NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A GENERIC SOFTWARE ARCHITECTURE FOR DECEPTION-BASED INTRUSION DETECTION AND RESPONSE SYSTEMS

by

Engin Uzuncaova

March 2003

Thesis Advisor:	James Bret Michael
Thesis Co-Advisor:	Richard Riehle

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: A Generic Software Architecture for Deception-based Intrusion Detection and Response Systems			**5. FUNDING NUMBERS**
6. AUTHOR(S) Engin Uzuncaova			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

Today, intrusion detection systems provide for detecting intrusive patterns of interaction. Although the responses of such systems are typically limited to primitive actions, they can be supplemented with deception-based strategies. We propose a generic software architecture combining intrusion detection and deceptive response capabilities in a uniform structure. Detecting and responding to attacks are realized via runtime instrumentation of kernel-based modules. The architecture provides for dynamically adjusting system performance to maintain continuity and integrity of both legitimate services and security activities.

14. SUBJECT TERMS Computer Security, Intrusion Detection, Intrusion Response, Deception, Software Architecture, Unified Modeling Language			**15. NUMBER OF PAGES** 85
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

A GENERIC SOFTWARE ARCHITECTURE FOR DECEPTION-BASED INTRUSION DETECTION AND RESPONSE SYSTEMS

Engin Uzuncaova
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1997

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

and

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2003

Author: Engin Uzuncaova

Approved by: James Bret Michael
Thesis Advisor

Richard Riehle
Thesis Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Today, intrusion detection systems provide for detecting intrusive patterns of interaction. Although the responses of such systems are typically limited to primitive actions, they can be supplemented with deception-based strategies. We propose a generic software architecture combining intrusion detection and deceptive response capabilities in a uniform structure. Detecting and responding to attacks are realized via runtime instrumentation of kernel-based modules. The architecture provides for dynamically adjusting system performance to maintain continuity and integrity of both legitimate services and security activities.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

This is the last part that I am adding to this thesis, so I am trying hard to capture what I really feel. Is it that I am happy and relaxed, because I have finally finished my thesis? I think that it is much more than that.

It has been a long journey. The point that I am at right now is not something that I imagined two years ago. I had the chance to know invaluable personalities and get their assistance throughout that journey: *Prof. J.Bret Michael* and *Richard Riehle*. I want to thank you for the inspiration, motivation and help that you gave me. If I am successful, it is not actually only mine, more like ours…

My wife has been always beside me, she even tried to learn C++ to help me when I was stuck! My parents always believed in me to pursue my goals. My son always tried to take me away from my work and wanted me to play with him.

To my son, Berk…

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

As military forces around the globe become ever more reliant on software-intensive systems for war fighting, the importance of these systems increases. Today, intrusion detection systems provide for detecting intrusive patterns of interaction; however, responses against attacks are limited to primitive actions such as terminating the intrusive processes and shutting down the system. As a result, valuable information about the attacks cannot be collected, which might otherwise be used to improve the security mechanism's capabilities to manage the exposure of information systems to the suspicious behavior of programs, with which they interact.

Deception-based strategies can be incorporated into intrusion detection and response systems to address to some extent the aforementioned weakness. The history of deception in military operations presents a wide spectrum of possibilities to realize deception in military information systems. We use deception as an active software component that interacts with intruders and provides fake responses to gather information about the nature of their interaction while protecting the key assets of the targeted system. The level of sophistication involved in those responses can be adjusted based on considerations such as system performance, the value of protected assets, the threat level of the attacks, and global security policies.

Intrusion detection systems are designed to discover intrusions on a computing system or the misuse of a computing system. However, most of the time these systems fail to differentiate between malicious actions of an attacker versus the egregious use of these resources by legitimate users. False positives and negatives along with monitoring overhead are problematic for intrusion detection systems, irrespective of whether they provide for *anomaly, misuse,* or *signature-based* detection. While some intrusion detection systems include response mechanisms, the capabilities of those mechanisms are constrained by the performance overhead introduced by both monitoring and response activities. Choosing simple responsive actions is considered to be sufficient as long as the detection mechanism does its job to detect attacks. However, with technological

improvements and creative thinking on the part of attackers, it is likely that more sophisticated deception techniques will need to be employed to successfully thwart cyber attacks.

Michael *et al.* [1,2] investigates the feasibility of applying deception in intrusion detection systems and introduced an abstraction, called intelligent software decoy, to demonstrate both theoretical and practical applications of deception in software-intensive systems. An intelligent software decoy is an abstraction for protecting objects within a component-based architecture from egregious and malicious use of their methods and interfaces. Deception is used to simultaneously make the intruders believe that they have been successful in accessing methods and interfaces, and permit the decoys to gather information about the intrusions. The concept covers a wide range of research topics including implementing military deception tactics in cyber space, developing deceptions, using kernel-based modules for intrusion detection and response capabilities, and developing a high-level specification language for specifying both detection and response rules.

Our focus in this thesis is on developing a deception-based intrusion detection and response architecture (DB-IDRA), with the intent to do the following: select an architectural representation for the DB-IDRA, and construct the architecture so that it can support a wide range of architectural frameworks.

The architecture that we settled on has the following characteristics. The architecture combines intrusion detection and deceptive response mechanism in a uniform structure. Responses are automated to minimize human interaction with the system. The DB-IDRA uses kernel-based software modules to detect attack sequences and respond to those attacks. To improve the survivability of the architecture, detection and response capabilities are distributed across the entire domain. To minimize the effect of defensive activities on the performance of the services provided by the system to legitimate users, DB-IDRA employs a mechanism to observe the global state and keep the system functioning within predefined tolerances. The architecture is based on a dynamic adaptation and evolution process to fulfill the abovementioned tasks. Global policy and doctrine are used to govern the behavior of the detection and response system.

The conclusions to be drawn from this research are mostly based on the architectural representation of a deception-based intrusion detection and response system. Architectural representation provides a consistent and comprehensive definition of the problem domain for future work that might refine or extend our findings. We proposed a means to incorporate deception into software-intensive systems as an active component. However, deception still presents complicated aspects that require more research regarding its applicability and feasibility in software domain.

THIS PAGE INTENTIONALLY LEFT BLANK

II. DOMAIN ANALYSIS

A. CYBER SECURITY

> Information warfare consists of those actions intended to protect, exploit, corrupt, deny, or destroy information or information resources in order to achieve a significant advantage, objective, or victory over an adversary. [8]

The preceding definition identifies the responsibilities of the information warriors. Information along with the computing resources has value to both their rightful owners and the rogue actors (a.k.a attackers). The owners can use either defensive or offensive means to protect their information and computing resources from being compromised. Likewise, the attacker will use offensive means to gain illicit access to these resources, while using defensive means in case the response of an owner is offensive in nature. In theory, defense can be more challenging than offense because defense entails protecting against all technically feasible vulnerabilities. Conversely, offense involves targeting specific vulnerabilities.

Information warfare has an impact on national security, from the well being of individual citizens and organizations (e.g., loss of confidentiality of tax identification numbers), to the security of nation states (e.g., compromise of the computers that control either the distribution of electricity within North America or the launching of intercontinental ballistic missiles (ICBMs)). As in kinetic-type warfare, there are laws, treaties, and other types of rules that place constraints on combatant actors' use of defensive and offensive tactics and strategies: violation of these rules would result in an actor being accused of war crimes. There are also laws and codes of conduct that pertain to the civilian actors in their use of defensive and offensive measures [5]. Although the "good guys" are often operating in a defensive mode, this is not necessarily the case. For instance, the Allies' victory in the Gulf War in 1991 was the culmination of their application of both defensive and offensive information warfare [9].

There is a wide spectrum of tools and techniques for conducting offensive information warfare. These may include simple social engineering ploys intended to trick or coerce legitimate users of computing resources into disclosing passwords or

cryptographic keys. Alternatively, they may be in the form of fully automated attack programs (e.g., malicious cooperating agents). On the defensive side, there exists a spectrum of tools and techniques for protecting computing resources, such as simple intrusion detection systems and systems that have an integrated detection-and-response capability. While defense against known tactics and strategies for attacks can be difficult to provide for, it is even more difficult to reengineer and apply defenses to address the continual emergence of previously unknown tactics and strategies of attack in a timely manner. This is part of the reason that conventional intrusion detection and simple response techniques have been ineffective against many real-world cyber attacks, such as the recent Sapphire worm [10]. Thus, as a defender, one needs some way of rapidly detecting new types (including variations on existing) attacks and deploying defensive measures to counter these attacks.

B. INTRUSION DETECTION

Intrusion detection systems (IDS) are designed to discover intrusions on a computing system or the misuse of a computing system, and often these systems fail to differentiate between malicious actions of an attacker versus the egregious (e.g., mistyping one's password or accidentally trying to run a user process in kernel space) use of these resources by legitimate users [1]. Either case may cause a denial of service to legitimate users (known as the false-positive problem). In an ever-changing world of cyber security, the defenders of computing resources often rely on IDS as a means of forward first line of defense. In general, there are two primary tasks associated with intrusion detection: *data collection* and *detection.* Some of the desired properties for an IDS, as listed in [11], are the following:

- ***Accuracy***: IDS must differentiate between a legitimate action in a system environment from an anomaly or a misuse, which relates to false positive and false negative issues.
- ***Performance***: IDS must provide sufficiently high-level of performance while not introducing an unacceptable level of overhead on the system being protected.
- ***Completeness***: While completeness emphasize that the IDS should not fail to detect an intrusion, this is very difficult in practice, because there is no global information about the attacks, which also includes past, present and future data.

- ***Fault tolerance***: IDS must be resistant to faults caused by the environment. Most of the time, IDS sit on top of commercially available operating systems and interact with third-party tools, potentially increasing the vulnerability of the overall system.
- ***Timeliness***: IDS must perform the analysis as quickly as possible. The countermeasures against a detected attack must be initiated before the attack would damage a system resource or IDS itself.

There are three typical approaches to intrusion detection: misuse, anomaly, and signature-based (a.k.a., specification-based) detection. *Anomaly detection* models normal or intended system behavior. Deviations from the model represent probable anomalies in the system [12]. There are challenges in this approach: defining what is "normal" behavior of a system and updating the model of normal or abnormal behavior for dynamic and complex environments. Anomaly detection can be used to detect previously unknown attacks, but is known to often result in high rates of false positives being generated.

In contrast, *misuse detection* defines behavior patterns and uses those specifications to locate any evidence of known attacks [13]. Specification rules are matched against the system audit data, thus, producing few false positives, but may result in a high rate of missed detections (i.e., events that should have been flagged as intrusions or misuses but were not). As the new types of attacks manifest themselves, the signature database needs to be updated.

The *specification-based* approach tries to address the weaknesses of misuse detection [14]. Instead of describing the events occurring in known attacks, a specification-based approach defines a program's intended behavior; then, deviations from that behavior can be flagged as suspicious behavior. This enables detection of previously unknown attacks and minimizes false positives. Additionally, with the retained precision of misuse detection, initiating immediate defensive actions is possible as violations are detected.

The shortcomings of existing IDS can be summarized under four different topics [15]:

- ***Variants***: Attack signatures are defined in response to known attacks; however, the difficulty here is that attack sequences are often changed easily and new exploits are defined over the known ones.

- ***False positives***: In defining the signatures or the behavior models to be used in IDS, the effort usually falls on the side of alerting too often rather than not enough. Filtering is a possible solution, but the risk of potentially missing an attack still stands as an open issue.
- ***False negatives***: A false negative occurs when the system allows an actual intrusive action to pass as a non-intrusive behavior. This can be more serious than false positives because it gives a misleading sense of security.
- ***Data overload***. This problem includes collection and analysis of the data. Collecting the right data and collecting the right amount of data results in a trade-off between performance overhead and effectiveness.

There are other challenging problems with current IDS such as identifying distributed attacks. IDS might fail to identify individual system events as malicious when analyzing the data from a single sensor. However a more comprehensive analysis of network activity could reveal an attack pattern, not only for a single network domain but also across domains.

C. INTRUSION RESPONSE

Increasingly, military and civilian information systems are becoming globally interconnected, thus tying their fates together (in terms of the effects of cyber attacks on any one system) on the security of the global domain. To address what might be called "composed vulnerabilities", a three-tier approach can be followed: *protect, detect, and react.* [16] The first two steps (protect and detect) have been explored extensively in the research community. For example, simple mechanisms such as encryption, firewalls, and authentication provide protection to some extent. Automated detection systems further filter message traffic for intrusions as mentioned in the previous section. Reacting to attacks is a relatively new subject focusing on techniques to prevent attacks from damaging the system and collecting information about the nature of those attacks.

Two primary reaction categories are introduced in [17]: *discrete administrative actions,* and *security policy reconfiguration.* Discrete actions include but are not limited to terminating connections, killing processes, and blocking certain communication channels. On the other hand, the second category (security policy reconfiguration) focuses on reducing the risk of further penetrations in dynamic environments, due to the fact that the systems usually are not designed with dynamic security in mind. Although this categorization presents different alternatives for intrusion response, these measures

are not always practical to deploy. Most of the time-responsive actions conflict with the intended system behavior and quality of service (QoS) because of the risk of deploying false and inappropriate responses.

Somayaji [18] introduces a delay-based intrusion response approach intended to defeat buffer-overflow attacks. This is accomplished by delaying suspicious system calls. However, this approach can adversely affect the level of performance and QoS for legitimate users. For example, in case of false positives, the delays could slow reaction or response time of the operating system to legitimate user requests. Furthermore, as the intensity of suspicious system behavior increases, the delay is more likely to introduce a performance overhead in general.

Survivable Autonomic Response Architecture (SARA) introduced by Lewandowski *et al.* [19] uses coordinated autonomic responses for defending information systems. Responses are generated automatically by a system without real-time human intervention, which rely on human knowledge and a policy that is programmed into the system in advance. As shown in Figure II.1, responses are selected based on either local or global system information. Global information is used by an orchestrator to effect coordinated responses, whereas local information is used to trigger local responses at the request of simple coordinators.

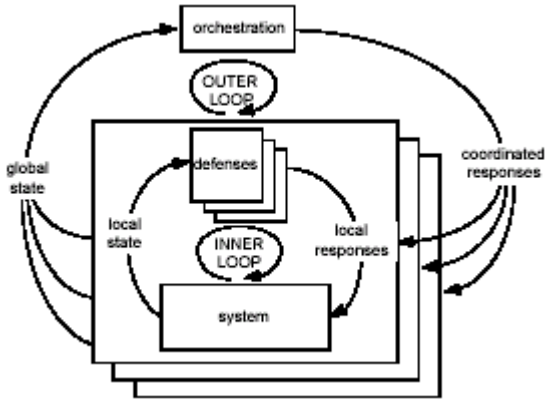

Figure II-1 The relationship of inner- and outer-loop responses (From: [19])

While the response systems using local information are able to respond quickly, taking global state into consideration increases the effectiveness of the response and contributes to the overall defense strategy. However, this kind of coordination depends on the efficiency of the analysis mechanisms used for response generation. Since the

analysis mechanisms are based on the system being protected and the potential threats to the system, that kind of response architecture should be adaptable to changing internal system structures and extensible to new components without disrupting the existing architecture. That is, the system should be able to endure modifications to underlying behavior without compromising the architecture or framework in which it resides.

Sekar *et al.* [20] emphasize the need for real-time reaction to attacks before they impact system performance or functionality. Figure II.2 shows how the detection engines generated by the offline components are used at runtime. Auditing Specification Language (ASL) enables specification of normal and abnormal behaviors of processes as logical assertions. Those assertions are based on the sequences of system calls and system-call argument values invoked by the processes. ASL specifications are compiled into optimized programs for efficient detection of deviations from the specified behavior. When discrepancies are detected at runtime, automatic defensive actions – also described in ASL – are initiated to contain or isolate the damage.

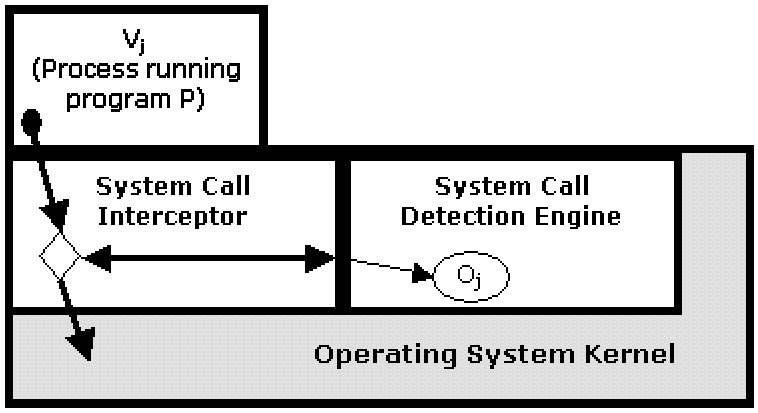

Figure II-2 Runtime system for execution of detection engines (From: [20])

The idea of moving both intrusion detection and response actions into the kernel is not new. Some of the advantages attributed to this approach are as follows: firstly, it reduces system overhead by avoiding extra context switching; and secondly, it is harder for an intruder to tamper with the IDS itself, as the attacker would have to modify the kernel to do so. However, there are also some disadvantages. Firstly, kernel-resident systems are not easily portable across platforms. Secondly, a misbehaving kernel module can do significant damage, because it has full access to the system [21].

For the most part, conventional response capabilities are limited to simple discrete administrative actions. The solutions providing more sophisticated techniques are prone to not be able to scale to large domains or adapt to dynamic environments.

D. INTELLIGENT SOFTWARE DECOYS

An intelligent software decoy is a software abstraction introduced by Michael *et al.* [1] for protecting objects within a component-based architecture from egregious and malicious use. In that regard, this paradigm for protection addresses all the steps in the three-tiered approach: *Protection* and *detection* mostly benefit from existing work, whereas *reaction* is based on deliberately deceptive responses. The main objective of this approach is to deceive attackers into thinking that their attacks have succeeded while simultaneously protecting the key components (or assets) of the system and learning about the nature of the attacks.

Deception-based approaches involve many levels of sophistication in terms of detection and response over that provided by conventional IDSs. Conventional IDS respond to attacks with primitive types of actions, which eventually indicate to attackers that they have been detected, likely causing the attackers to change their strategies and tactics: This leaves little if any opportunity for the defender to learn about the attack strategies and tactics. In contrast, automated deception-based responses can be used to engage the attacker for as long as possible to gather information about the nature of the attack (e.g., behavior pattern, impact on the system, and origin). Deceptive responses can be automated much like the attacks. Such responses can be effective because attackers depend on the honesty of the computer systems they attack. Deception can confuse the attackers' planning or frustrate them for a while without indicating that their attack has been detected. This could be especially important during intensive information warfare when terrorists attempt to bring down critical systems in a short period of time: Delaying intrusive processes gives the IDS time to analyze the attack and plan a response. Deception also allows for turning an attacker's own strengths of patience and determination against the attcaker, much as is done in the defensive martial art known as *Aikido* [37].

Michael *et al.* [2] proposed a high-level architecture for software decoys, shown in Figure II.3. The architecture is based on instrumentation of kernel libraries and log

monitors available on a chosen platform. The basic instrumentation provides for the detection of events and event attributes specified in the generic behavior model and the interface for executing monitoring programs. A specialized compiler will generate monitoring programs from a high-level language, such as CHAMELEON [3], for specifying both detection and response rules for carrying out deceptions. Below is a rule combining an event pattern with an action, represented in CHAMELEON. The rule specifies that each time a read event is detected, and the buffer contains the string "SITE EXEC", then the value "NOOP" should be assigned to the buffer. This rule is used in a case study, reported in [3], to detect and respond to an attack against Washington University's ftp server (wu-ftpd), which uses the `SITE EXEC` command to gain root privileges [42].

detect x: read & post (buf(x) == "SITE EXEC")
from execute-program do buf(x) = "NOOP"

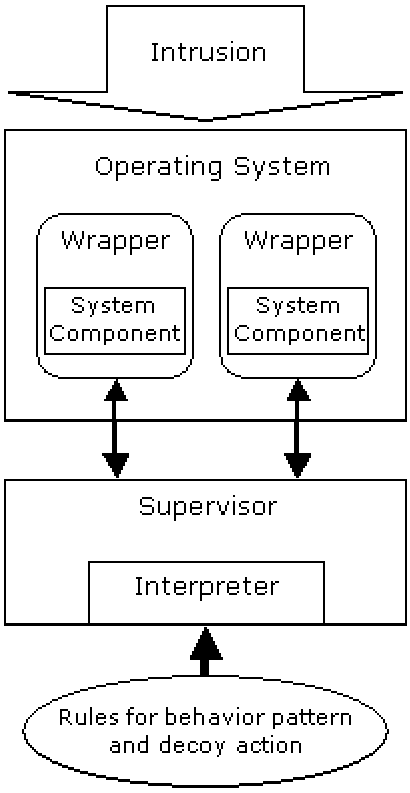

Figure II-3 High-level software decoy architecture (From: [2])

In the software-decoy architecture, the supervisor coordinates the actions among the decoyed-enabled components in order to make decisions on how best to instrument

components given the latest information about component interaction and selected responses – conforming to information operations doctrine and policy – to effect system-wide deception strategies. In that regard, the decoy approach has in common some of the founding goals of the SARA architecture (described in the previous section), such as rapid and coordinated responses to patterns of suspicious system events.

Software components are wrapped with decoy functionality on a selective basis: Wrapping can be performed at more than one level of abstraction from application-level objects such as web applets to low-level operating system calls. This provides a level of flexibility in coordinating the overall system strategy dynamically as characteristics and intensity of threats change. If any of the assertions (e.g., preconditions, postconditions, or invariants) built into the behavioral model for the system are violated, the software decoy isolates the interaction with the intrusive process into an *antechamber* to further analyze the process' real intention and apply, if necessary, deceptive responses to gather information. The antechamber is either hosted by the operational system on which the software decoy resides, or on a separate processor or platform in order to minimize the effect of the monitoring and decoy actions (*e.g.*, those of delay tactics) on the availability and performance of computing resources requested by legitimate users of the decoy-enabled software components.

Figure II-4 illustrates a possible scenario, in which the protected system is under a buffer overflow attack. The main components in this diagram are the wrappers, supervisor and repository of detection rules and decoy actions. The software component is wrapped by decoys with the purpose of both detecting buffer overflow attacks and responding back to those attacks with system delays. When a malicious request is made to the component, which violates the contract specifications, the buffer-overflow-wrapper passes the intrusion/misuse information in the form of event traces to the supervisor. Then, the supervisor analyzes the situation based on the rules included in the repository and selects a response strategy. This strategy should conform not only to local scope, but also to the global system state and the defense strategy. Based on the analysis results, the supervisor invokes pre-programmed decoy actions in the wrappers and installs new wrappers to change the granularity of monitoring if necessary. In this particular scenario,

the supervisor invokes a wrapper to delay the malicious process for a finite period of time, possibly determined using some form of counter planning [7].

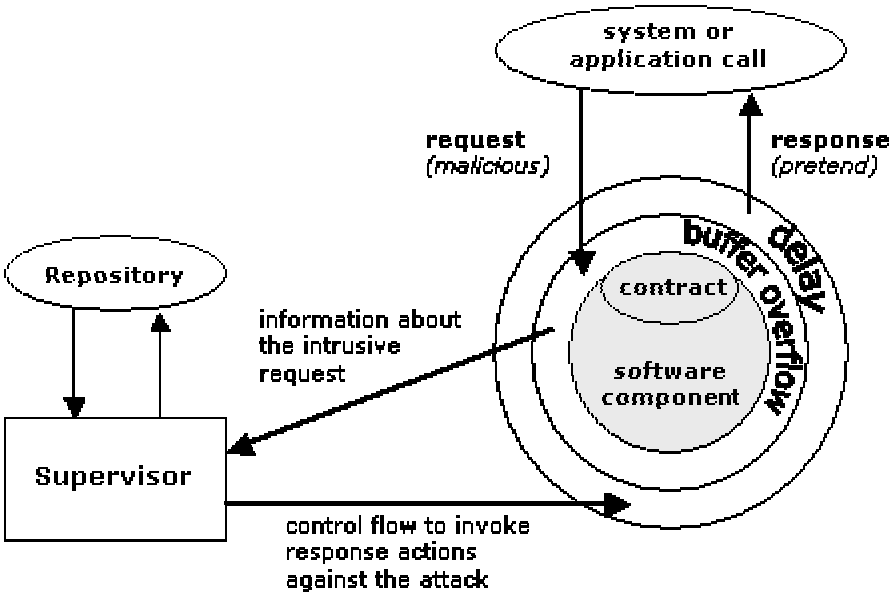

Figure II-4 Decoy interaction with a buffer overflow attack (After: [6])

The salient features of the decoy approach can be summarized as follows:

- ***Deception as a response strategy.*** Traditional warfare and military tactics presents many opportunities that can also be applied in the cyber world. Deception also increases the range of responses from the production of a fake error message to the simulation of the whole operating system. Another benefit is collecting information about the attacks, which enables better preparation for future attack scenarios.
- ***Dynamic configuration.*** As the context changes, it is possible to update the structure of defensive components for both detection and response purposes and reconfigure the overall strategy according to the decoy policy and doctrine.
- ***Automated response.*** Unlike most of the intrusion detection systems, software decoys integrate detection and response mechanisms within a single automated framework. This reduces the need for human intervention at runtime. While having some disadvantages, kernel-based detection and response provides fast and efficient mechanisms to confront the attacks before they potentially damage the system.

III. DEVELOPMENT PROCESS

A. OVERVIEW

Software-intensive systems require an organized approach for the creation, analysis, and maintenance of architectural design. An architectural design is defined in terms of architectural abstractions, such as patterns, styles and views. The way architectures are designed has been evolving towards a more challenging and complex process, because mapping the real-world problems into a software domain requires more and more multifaceted considerations, particularly related to complexity and new technological developments. To overcome this challenge in system development, architectural design provides a transition from the user world (e.g., domain, requirements and risk analysis) into software-related abstractions (e.g., software components, connectors, interfaces) by describing the elements of the system and how they work together to fulfill the system's requirements.

Architectural design activities are usually separated into different *views*. Each view addresses one or more of the concerns of the system and the stakeholders; therefore, a view refers to the expression of a system's architecture with respect to a particular set of conventions by which a view is created, depicted, and analyzed [22]. Although there is no common approach in separating architecture into different views, the main reason behind multiple views is to be able to manage complexity.

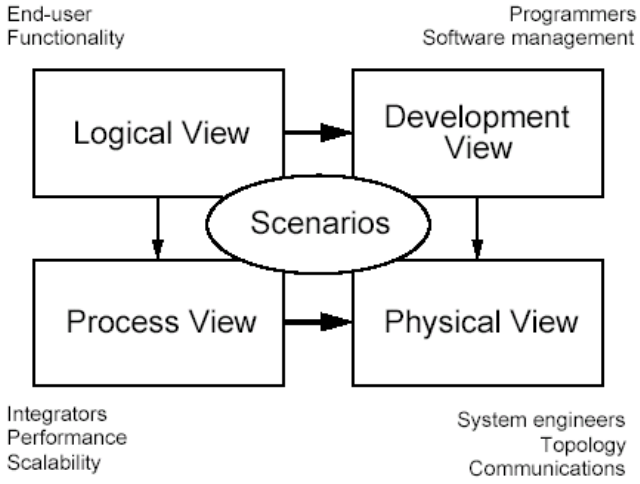

Figure III-1 The "4+1" view model. (From: [23])

In the literature, there are several approaches explicitly treating different views of architecture. The 4+1 approach, by Kruchten, describes software architecture using five concurrent views: *logical, process, physical, developmental,* and *scenarios.* As Figure III.1 shows, each view addresses a specific set of concerns of interest to different stakeholders in the system [23]. Hofmeister *et al*. used a four-view model, as shown in Figure III.2, describing the architecture from four different structures: *conceptual, module, execution,* and *code*. The four-view model is the result of a search for commonalities across domains and underlying principles that lead to good and useful software architectures [24].

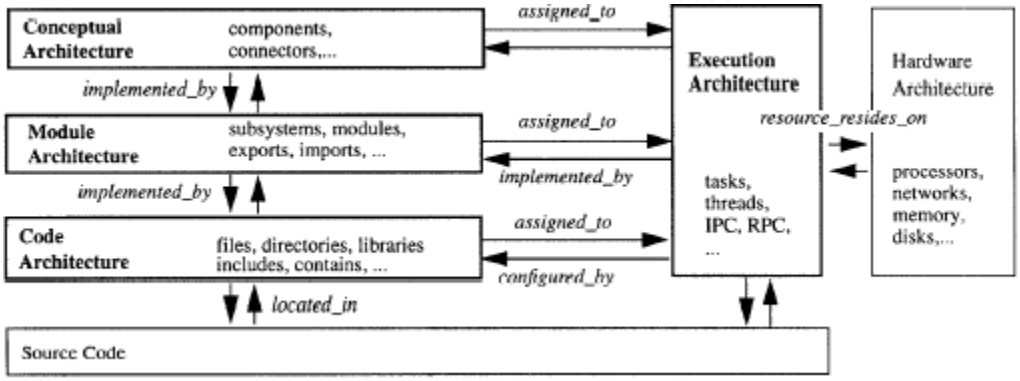

Figure III-2 The four views of software architecture (From: [43])

The main goal of architectural views is to articulate different aspects of the system in a loosely coupled and highly cohesive way. Considering that principle, both of the models provide a tenable solution to the problem, while preserving their distinct characteristics. However, the essence of using a view model is that it should provide such a level granularity that the designers could define the system properties clearly within the scope of each view. Most of the time, achieving such granularity is hard due to the complexity of a problem domain. That is why, existing view models may need to be modified according to the characteristics of the problem. For example, an architecture for a web-based shopping application presents different characteristics than an application in the computer security domain.

For intelligent software decoys, the problem domain is not fully explored yet. For example, incorporating deceptive responses in an intrusion detection framework is a relatively new subject, and the effects of both defensive and offensive deception on a

cyber domain are not well-understood. Our architectural design process is intended to provide a generic framework to be used to demonstrate the feasibility of deception-based intrusion detection and response approaches. For the required level of abstraction, we want to explore the domain in two views: *conceptual* and *module*. The conceptual view enables us to explore the concept, whereas the module view allows us to map the conceptual view to the actual problem in a form of architectural description. Besides these two architectural views, it is possible to enhance our view of the system by decomposing the problem into its major subcomponents: *monitoring, intrusion detection, intrusion response, analysis,* and *planning.* Figure III.3 represents the architecture views based on both the architectural abstractions and major subcomponents of the system.

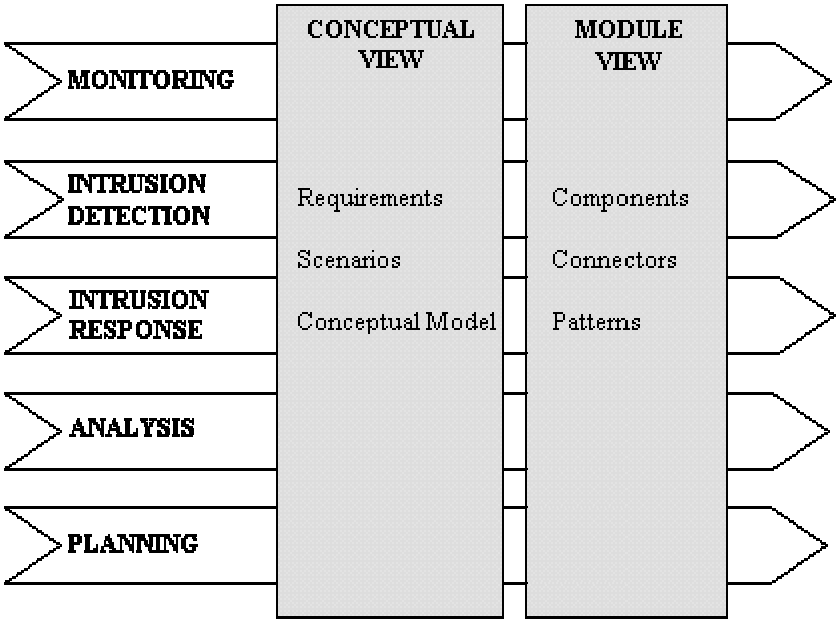

Figure III-3 Architectural views for intrusion detection and response framework

The decomposition of the entire problem domain allows us to investigate sub areas independent of each other. This introduces a simplified view of the system. *Monitoring* includes data collection and observation tasks. Based on these data, *analysis* provides the required decision-making capability for both *intrusion detection* and *response* mechanisms. On the other hand, *planning* involves the required set of rules for governing all the other tasks. Planning uses both low-level rules defining atomic actions

(e.g., response actions, detection patterns) and high-level rules defining the global defense strategy (e.g., policy, doctrine).

B. DEVELOPMENT PHASES

Developing the architecture for a deception-based intrusion detection and response framework is a complex problem. Major steps in this development process are understanding the problem, exploring the domain for possible approaches, finding and elaborating the requirements, conceptualizing the problem, mapping the conceptual view into architectural abstractions, and developing the detailed architecture. Regardless of its complexity, it is possible to solve the problem in a "grand-design" approach; however, there are some significant risks associated with this approach. A better way is to separate the development into phases, and follow a incremental iterative approach such as the Rational Unified Process [39]. For this purpose, we use a three-phase process to develop the architecture defining the activities from requirements through to detailed architectural design: *plan and elaborate, high-level system definition,* and *detailed architectural design*.

1. Plan and Elaborate

The *Plan and Elaborate* phase includes the initial concept exploration, investigation of possible approaches, planning, specification of requirements, and developing use cases. At the end of this phase, we expect to define "what problem we are trying to solve." Understanding the problem is essential, because the requirements and the design will be built on top of this definition. Then, the fundamental requirements will be expressed and scenarios will be developed to further elaborate these requirements. This phase will generate the *functional* and *non-functional requirements* and the *use cases*.

2. High-Level System Definition

The *High-level System Definition* phase uses the artifacts generated in the previous phase and starts building the system. The essence of this phase is to leave out the low-level design details and focus on the high-level aspects of the system, including major components and their collaborations. The c*onceptual model* is the primary artifact of this phase, which decomposes the problem into meaningful concepts and shows their

collaboration. Another artifact is the *high-level architectural design*, which maps those concepts into architectural abstractions, such as components and connectors.

3. Detailed Architectural Design

The *Detailed Architectural Design* phase defines the architecture in a sufficient level of detail including subcomponent, interface, and system behavior specifications. The essence of the detailed architecture is that it integrates the functional and non-functional requirements into the system structure. Although the implementation details are excluded at this stage, the architecture should provide an unambiguous approach, which will support implementation-level activities later on.

C. A MODELING TOOL: UML

> The Unified Modeling Language (UML) is a standard language for writing software blueprints. The UML can be used to visualize, specify, construct, and document the artifacts of a software intensive system [25].

The role of software architecture in software-intensive system design is important and, as those systems become more sophisticated and larger, architectures become increasingly essential for understanding, managing and describing their complexity. While the benefits of architectural design are obvious, there are several shortfalls in the representation of architectures [26]. The ad-hoc and informal approaches to architectural design present several significant challenges to be overcome.

- Architectural designs are often poorly understood and not amenable to formal analysis or simulation.
- Architectural design decisions are based more on default than on solid engineering principles.
- Architectural constraints assumed in the initial design are not enforced as the system evolves.
- There are few tools to help the architectural designers with their tasks [27].

In an effort to address these challenges, a level of formalism is required for describing architectures, and therefore, supporting overall software development lifecycle efforts. That level of formalism can be achieved through various ways, including programming languages, module interconnection languages, interface definition

languages, and architecture description languages (ADL) [28]. Approaches other than ADLs fall short in representing architectural abstractions for several reasons:

- They are defined to represent low-level constructs, which is not adequate for architectural abstractions.
- They cannot represent reusable architectural patterns.
- They address specific problem domains; therefore, extendibility of those methods is low.
- They mostly focus on the static structure of the systems, and hence, the dynamic behavior of systems cannot be described sufficiently.

On the other hand, ADLs are used to define and model system architectures prior to implementation. Besides static structures of systems, they also address system functionality, interactions between components, and interfaces. There are numerous ADLs, such as Rapide, Wright, ACME, UniCon, and MetaH. Detailed information about ADLs can be obtained from [29].

The capabilities of UML are limited when compared to ADLs. However, the rich set of tools provided by UML supports the representation of software systems. The main advantage of UML is its wide acceptance within the software engineering community: It provides a lingua franca for communication among designers, architects, and other. While the notations in UML present many alternatives in describing the implementation of a system, using the same kind of notations in architectural design blurs the distinction between implementation and architecture views. For example, UML provides only interaction diagrams for describing the dynamic properties of a system and using these diagrams in both implementation and architecture views may create confusion.

Considering the pros and cons, UML is well suited for the purpose of our study. Our main goal is to be able to represent the architecture in an understandable way. The power of UML for representing the conceptual design and static structure of a system allows us to visualize our ideas in an efficient and organized way. UML also provides "lightweight extension mechanisms" – stereotypes, tagged values, and constraints – that can be used to enhance the language with architectural definitions. Further discussion of the applicability of using UML to describe software architectures can be found in [30] and [31].

IV. PLAN AND ELABORATE

A. OVERVIEW

Plan and elaborate is the first step in the developments process and the main purpose of this phase is concept exploration. The requirements produced in this step constitute the foundation for architectural design activities. We explored the domain in two ways: by capturing the high-level requirements and then modeling those requirements with use cases to gain a better understanding about the system. This phase produces the entities involved in the system and the interactions between those entities.

1. Requirements

Requirements analysis, for our problem domain, covers the high-level functionality and associated non-functional aspects of the system. We used two primary resources for requirements elaboration: project group meetings and related literature. In that regard, we explored the computer science literature in related areas such as intrusion detection approaches, runtime code instrumentation, kernel-based software modules, and deceptive strategies.

The set of requirements, developed in this phase, represent the key functional and non-functional features of the system. Without going into implementation detail, we defined what is really needed for the system in a clear form. Requirements are generally described in textual format. They fall short in visualizing the relations between the system and its users. For that reason, we augmented the requirements with use cases, in which we were able to make readily visible the fundamental behavioral aspects of the system.

2. Use Cases

Every system consists of a set of sub components and a set of interactions either internally or externally. Use cases provide a view to describe the behavior of the system as seen by its end users. This view focuses on the aspects that outline the system's architecture. Use cases are based on the actors that the system interacts with, the roles of these actors, and how they interact with the system. Additionally, use cases specify the behavior of a system or a part of a system, and are descriptions of a set of sequences of actions, including variants, that a system performs to yield an observable result or value

to an actor [25]. In that regard, we developed a use case model to capture the intended behavior of the system and augment the requirements.

B. FUNCTIONAL REQUIREMENTS

1. Intrusion Detection

Deception-based intrusion detection and response architecture (DB-IDRA) combines both intrusion detection and response in a single structure. As described in section II.B, there are three common techniques to intrusion detection: *anomaly, misuse,* and *signature-based.* Although there are specific concerns associated with each technique, DB-IDRA provides a generic framework in which any intrusion detection approach can be integrated. The point is that the intrusion detection and response mechanisms should be able to communicate with each other as defined by the architecture. In any case, the architecture should specify the necessary components along with the data structures.

2. Deception-based Response

DB-IDRA uses a deceptive defense strategy for intrusion response. The main objective in this approach is to be able to deploy deceptive tactics against intrusions to gather as much information as about the nature of the attack. The spectrum of possible tactics needs to cover a wide range from mimicking of normal system behavior through inventing of fake activities to attract the attacker's attention. The appropriateness and effectiveness of the deceptive tactics depends on the situation, including the intensity of conflicts, the value of the information being protected, and the threat level of attacks. Additionally, deception strategies should not introduce an unreasonable performance overhead on the system.

3. Automated Response

DB-IDRA integrates the automated response functionality in the architecture to realize deception-based defense strategies. It should be capable of making decisions and performing actions that help the system accomplish its mission much more quickly and accurately than a human could. Capabilities for specifying automated response actions must be invoked as intrusions are detected, which provides damage prevention and containment. This functionality also minimizes the constant involvement of human experts in runtime activities.

4. Runtime and Offline Analysis

Combining intrusion detection and response in a uniform framework increases the performance overhead due to the complex decision-making process. The analysis mechanism is grouped into two layers: *runtime* and *offline.*

- *Runtime analysis* supports the decision-making capabilities for detecting and responding to attacks, and maintaining an acceptable level of system performance. The main concerns are: (i) fast and efficient detection with false positive rates, (ii) maintaining the consistency of the defense strategy against the global policy and doctrine, and (iii) maintaining the system performance by observing the system state.
- *Offline analysis* involves analyzing the past data and improving the system for detection and response approaches (e.g., addition of new response actions and new attack specifications), high-level policy and doctrine rules (e.g., switching from defensive to offensive countermeasures), and system functionality (e.g., addition of the capability to trace the origin of attacks).

5. Kernel-based Detection and Response Modules

Kernel-based operations avoid overhead caused by extra context switching. However kernel-resident implementation introduces a level of complexity in management and configuration of entities inside the kernel and also has an impact on the host behavior. These problems can be addressed by a careful design approach. DB-IDRA uses kernel-based modules to detect attack sequences and respond to those attacks. It should also provide means to modify these modules dynamically. Ko *et al*. discusses the feasibility and practicality of in-kernel intrusion detection and introduced the Generic Software Wrapper Toolkit as the basis for implementing kernel-resident intrusion detectors in [32] and [33].

6. Distributed Detection and Response Capabilities

DB-IDRA should distribute detection and response capabilities across the entire domain to eliminate single-point of vulnerability. The advantage here is that doctrine and policy developed for systems could be integrated into larger joint information operations and be distributed throughout the cooperative engagement grid [6].

7. System Performance Maintenance

While protecting a system against intrusions, performance overhead could degrade the level of service provided by the system to legitimate users. DB-IDRA will include a mechanism to observe the global state and keep the system functioning within

predefined tolerances. For example, as the intensity of conflict increases, sustaining interactions requires more system resources and eventually causes degradation of system performance. In order to address this problem, a set of system variables can be monitored at runtime, such as the load of malicious interactions on the network. If this load is beyond a predefined threshold value, then necessary measures would be taken to keep the system stable.

8. Dynamic Adaptation and Evolution

Dynamic environments require a system to adjust its behavior at runtime. This kind of an adjustment can be achieved in two successive steps:

- Observing the system's behavior and analyzing the observations to determine appropriate adaptations, and
- Carrying out these adaptations in such a way that the system protects its consistency and integrity.

DB-IDRA should include such mechanisms to provide addition, removal, or replacement of components (e.g., adding new sensors), and modifications to the configuration of components (e.g., response actions and defense strategies).

9. Global Defense Policy and Doctrine

Global policy and doctrine are required to govern the behavior of the detection and response system. Policy and doctrine represent a set of high-level rules indicating essential properties (e.g., appropriateness, limitations, course of actions, and so on) that a defensive strategy should conform to. For example, an administrative domain may be bounded to specific legal regulations and is able to deploy only defensive actions against adversaries, whereas another domain may extend its defense strategies to be more offensive. This also indicates that these rules are dynamic in nature and can be modified according to the domain-specific scenarios.

C. NON-FUNCTIONAL REQUIREMENTS

1. Interoperability

Interoperability, within the context of intrusion detection and response systems, represents the need for effective communication and collaboration among different administrative domains. While sharing the common purpose of protecting their domains via efficient mechanisms, those administrative domains are independent from each other. What they need is a strong collaboration that will allow them to update their own security

measures and inform the other domains about ongoing security-related events and their results. For an effective communication, timeliness and security of the information is critical. The ultimate goal of interoperability is to provide the necessary means to establish this collaboration scheme.

2. Stability

Stability, in general, is the property of a system ensuring that the system will remain within defined and recognizable limits against disturbances. Resilience is another concept defining stability in two dimensions: (i) how a system behaves in order to maintain stability within a specific stability domain (i.e., engineering resilience), and (ii) the intensity of disturbances that can force a system into a different stability domain (i.e., ecological resilience) [34].

Based on these two definitions, stability of an intrusion detection and response system mainly focuses on the disturbances from the environment, and the defined and recognizable limits for the system behaviors. Besides defining these key elements for the system, it is also necessary to evaluate the stability under varying levels of intensity of conflict to guarantee the system operates as intended.

3. Scalability

DB-IDRA represents a total defense strategy consisting of single host to network-wide and inter-domain protection. Implementing this hierarchical strategy requires a scalable architecture. As the size and complexity of the domain increases, the intended use of the system must behave as defined and critical system properties must be preserved. For example, logical groups of individual hosts may implement a common deception strategy and also need to cooperate with a different administrative domain. One of the requirements for that strategy is for a central unit to manage the global defense strategy and inter- and intra-domain communication paths.

4. Survivability

DB-IDRA has a critical mission to protect software systems against attacks in bounded time. In the presence of threats against the system, DB-IDRA must be able to remain operational throughout the mission. This is the issue of survivability. Furthermore, mechanisms must not introduce additional vulnerabilities that can be exploited to degrade the survivability of the system.

In the first case, survivability scope is limited to the incidents caused by attacks. However, survivability is also subject to vulnerabilities caused by failures and accidents that are not necessarily related to attacks. Policy decisions related to survivability can change continuously according to system context; it is even possible to observe evolution of survivability in a particular system as problems occur and are handled, mission objectives change, and the intensity and load of the environment changes.

The second case is the focal point for maintaining DB-IDRA's own survivability. Essential functionalities and properties in decoy mechanisms must be maintained in order for providing continuous protection. Even in case of failures or attacks, DB-IDRA must behave in accordance with the mission objectives. Distributed capabilities can address this issue to some extent.

5. Effectiveness

Protecting a system from cyber attacks is a challenging task in its own right. Combining this task with a sophisticated response mechanism makes this task even more challenging. This combination requires a set of rules for both detection patterns and decoy actions. Representation of these rules requires the use of a sufficient level of formalism, as the rules must be precise and concise in their representation. Even small ambiguities in rule definitions may cause some security holes in systems; that is why, a continuous evaluation of the general system state is necessary for the effectiveness of the system. It is also possible that a set of rules considered to be efficient in one scenario may turn out to be inadequate in another scenario. New types of threats to information systems introduce the need for new measures to be taken, as well.

Effectiveness has two different views regarding the mentioned facts: providing effectiveness and maintaining effectiveness. Although it is necessary to provide an effective mechanism to facilitate the desired defensive strategy, situations such as those mentioned above could force a change of context that forces the system to operate ineffectively. Evaluation of the system state is, therefore, important to locate these inefficiencies. The system should also provide a set of corrective actions to recover from those ineffective states.

6. Performance

Performance is a general constraint over all the functional and non-functional requirements. Means must be provided to minimize the affect of the monitoring and decoy actions (*e.g.*, those of delay tactics) on the availability and performance of computing resources requested by legitimate users of the protected software components.

D. HIGH-LEVEL USE-CASES

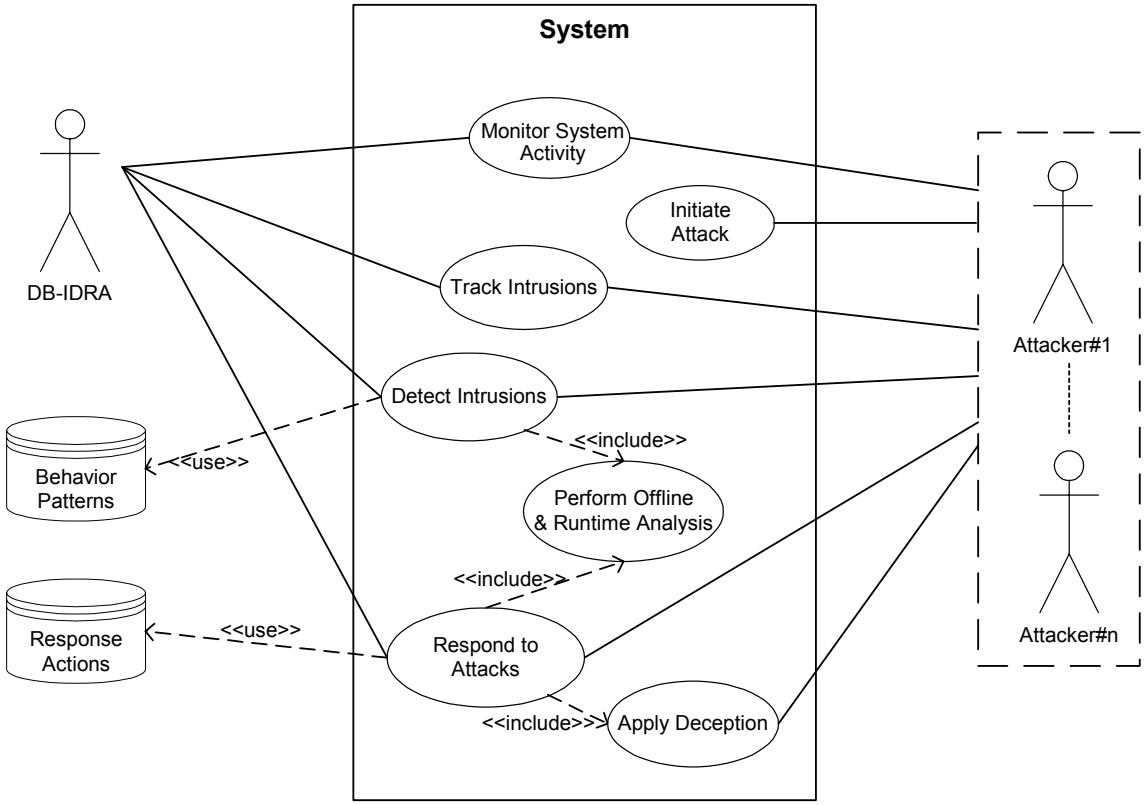

Figure IV-1 High-level use case diagram for DB-IDRA

Figure IV-1 shows the use case diagram for DB-IDRA. The diagram includes the fundamental system functionality such as monitoring, intrusion detection, deceptive responses and analysis capabilities. Each ellipse represents an individual use case describing a set of actions that the system performs to yield an observable result. Stick figures represent the roles that users play when interacting with the use cases. The associations between the actors and use cases show that they communicate with one another, possibly sending and receiving messages. While use cases can be organized by specifying generalization, include, and extend relationships, in this specific use case diagram, there are only include relationships between use cases. This type of relation

means that the base use case (e.g., respond to attacks) explicitly incorporates the behavior of another use case (e.g., apply deception).

1. Initiate Attack

Actors : Attacker(s)

Type : Primary

Description : DB-IDRA protects a software component from cyber attacks. Therefore the basic assumption is that there are intrusive actions against the protected entity. Not only individual attackers initiate those intrusions, but also organized groups conduct attacks on systems depending on their value, and criticality. The complexity of those attacks can range from simple scripts to distributed attacks. Those attacks usually try to invoke unauthorized queries and attempt to modify data or processes. While the origin and purpose of attacks are important to some extent, intensity of attacks presents another significant problem for counter-measures.

2. Monitor System Activity

Actors : DB-IDRA, Attacker(s)

Type : Primary

Description : Monitoring is the first step in observing the activities on a system. Monitoring is used to observe events and system parameters for intrusion detection and system performance purposes. Monitoring usually perturbs system performance by consuming system resources such as CPU or storage. Minimizing this overhead is, therefore, necessary for an efficient real-time intrusion detection and response framework. Monitoring is divided into layers to analyze various components involved in monitoring process: *observation, collection, analysis, presentation, interpretation, console,* and *management* [35]. For our problem domain, monitoring covers the first two layers in this sequence. The observation layer gathers raw data on individual components of the system, and the collection layer collects data from different observers. This approach introduces a hierarchy of sensors focusing on different aspects of the system being monitored. Besides an organized structure, this approach also allows for systematic use of monitoring, such as turning off some observers, modifying some aspects of the observers, and reconfiguring the hierarchy.

3. Perform Runtime and Offline Analysis

Actors : DB-IDRA

Type : Primary

Description : Analysis is performed at two levels: runtime and offline. Runtime analysis is done for evaluating the system state and making decisions about intrusions and responses. Runtime analysis governs the dynamic behavior of the system. On the other hand, offline analysis works on system audit that is recorded at runtime and evaluates how well the system performed its mission. This evaluation concentrates on the performance and effectiveness of the system and also finds deficiencies, abnormalities, or suspicious events, if any, related to intrusion detection rules and response actions. Evaluation results can be used as a means for process improvement.

4. Detect Intrusions

Actors : DB-IDRA, Attacker(s)

Type : Primary

Description : Every attack follows a specific behavior pattern, and DB-IDRA employs an intrusion detection mechanism to match behavior patterns against a central shared database of detection rules at runtime. DB-IDRA introduces a level of flexibility to use any kind of detection technique. While some simple attack sequences can be detected at the observation layer (i.e., decoy wrappers), more sophisticated attacks may require further system- and network-wide analysis to draw a final conclusion about the nature of the interactions.

5. Track Intrusions

Actors : DB-IDRA, Attacker(s)

Type : Primary

Description : Once an intrusion is detected in the system, the response mechanism isolates it from the system into a different mode. The distinction between different operating modes is necessary to keep track of deception-related activities. As introduced in [1], when an intrusion is detected, the software decoy switches from its nominal operating mode to a deception mode for that particular intrusive process. In the

deception mode, the interaction with the process is redirected to a construct called an *antechamber*. The antechamber serves as a waiting area for the requests initiated by intrusive processes. The software decoy assesses the nature of the attack and generates responses while the process is kept in the antechamber. Both attack attempts and egregious actions may trigger that transition between operating modes.

6. Respond to Attacks

Actors **:** DB-IDRA, Attacker(s)

Type **:** Primary

Description **:** DB-IDRA includes both intrusion detection an response functionality. The response mechanism requires an efficient detection approach. Ranging from simple response actions such as terminating the connection, to sophisticated responses like deception, there is a wide range of possibility to realize responsive strategies. The complexity of responses is likely to increase the overhead for the system. Therefore, feasibility of response approaches turns out to be important. Responses are performed by kernel-based wrappers on a selective basis. Criteria for a response must be related to the defense policies, system state, and the characteristics of each particular attack.

7. Apply Deception

Actors **:** DB-IDRA, Attacker(s)

Type **:** Primary

Description **:** Unlike many primitive intrusion responses, deception is a sophisticated approach against cyber attacks, which aims to fool an intruder into believing the attack is successful while both gathering information about the nature of attacks and protecting the key assets of the system. Some example deceptive responses include fake error messages, delays in responses, and lying about the status of the files [2]. Feedback about the effectiveness of the deception tactics is also important for a good defensive deception.

V. HIGH-LEVEL SYSTEM DEFINITION

A. OVERVIEW

High-level system definition is the second step in the development process for the architectural design of DB-IDRA. This phase is the bridge between initial concept exploration and detailed architectural design. This phase takes the artifacts defined in the previous step (i.e., requirements and use cases) and produces the conceptual model of the system and high-level architectural design, which, in turn, become the input to the detailed design phase.

The conceptual model illustrates the concepts in the problem domain and helps understand the problem domain further. The concepts represent the essential entities that will eventually act the way in the real world. The conceptual model presents the problem domain, abstracted out from design details. Formally, a concept consists of its *symbol, intension* and *extension* [40]. A concept should have a specific name, a context, a definition, and a set of examples to which the concept applies. Associations between the concepts show how concepts relate to both the model and other concepts. Typically, it is the symbol and intension of a concept that is of practical interest. The intension defines the concept with an emphasis on the context that applies to the concept. Modeling starts with identifying the concepts, and this can be done via two different techniques. The concept category list can be used to provide a list of candidate concepts related to different categories, such as physical objects, places, events, and transactions. Another approach is based on noun-phrase-identification technique, that is, to identify the noun and noun phrases in textual descriptions (e.g., requirements, use cases) of a problem domain. Our approach used both techniques to come up with an inclusive list of concepts as described in the following section.

Software architecture is defined as a structural plan that describes the elements of a system, how they fit together, and how they work together to fulfill the system's requirements [24]. In order to describe the architecture as software abstractions, there is a need to map requirements to architectural entities. The high-level system definition combines the artifacts from requirements analysis and conceptual modeling and produces

the main entities and the structural organization of the architecture. In phase, the main entities are the logical groupings organizing functionally related elements of the system. For this study, the main product of high-level design activities is that the architecture is decomposed into layers consisting of architecture packages with distinct functionalities.

B. CONCEPT CATEGORY LIST

Generation of a concept category list is an intermediate step to find and describe the candidate concepts for the architecture's conceptual model. The concepts are defined by their names and intensions. The set of concepts provides for building the domain vocabulary and dealing with the complexity at simpler levels of abstraction by using decomposition. The following list describes the concepts extracted from the requirements and the use cases for DB-IDRA.

Administrative Domain	Administrative domain consists of a number of hosts (systems). Domain also represents the entity for which the decoy policy is in effect.
Host	Host is an individual computer system that includes software components to provide services to its users.
Software Component	Software component is either a complete program or a sub program. Software components provide services to system users.
Contract Specification	Contract specification represents the formal definition of component interfaces. Contract specification consists of three sub elements: precondition, postcondition, and class invariants.
System Activity	System activity stands for all the interactions taking place in a computer system.
System Event	System activity is described by sequences of system events (event traces) initiated by the processes in that system. Events perform some actions that change the system state.
System State	System state represents the combination of system variables and ongoing activities at any given time.
System Metric	System metrics, within the context of software decoys, define the nominal system state so that it is possible to measure system state against those metrics and determine corrective actions.
Wrapper	Wrappers are kernel-based constructs that interact with the processes in the system. They are deployed and coordinated by the supervisor. They perform the actions that are assigned to them.
Probe	Probes collect data about the system state, and pass those data to the analysis component. They are different from the wrappers in a sense that they do not participate in the defensive actions.
Intrusion Detection	Intrusion detection refers to the se of monitors to recognize intrusive behaviors. Generally, intrusion detection includes response handling, but this study intrusion response is considered separately from detection.

Intrusion Response	Intrusion response refers to actions that are taken after attacks are detected. In traditional approach, responses address containing and recovering from damages and hardening defenses. DB-IDRA extends this scope to also include deception-based defense strategies.
Behavior Pattern	Behavior pattern represents sequence of system events. Depending on the detection technique deployed in a system, patterns define either nominal system behaviors, or attack sequences. Each behavior is defined by unique sequence of system events.
Intrusion	Intrusions are either malicious (intentional) or egregious (unintentional) use of the system resources. Intrusions ask for system resources in a sequence, thus, their behavior represents a pattern.
Attacker	Attacker is the person who initiates intrusive processes against a system. Attackers can be either individuals or coordinated groups. Organized attacks can be more sophisticated than amateur individual attacks. While persistency, and motivations are different for each type of attacker, there are commonalities in basic attack techniques that are employed.
System Action	System actions optimize the system performance.
Decoy Action	Decoy actions are the embodiment of the defense strategy and they are specified in the software wrappers.
Action Specification	Both system and decoy actions are specified in a high-level specification language.
System Log	System log stores defense related system activities and is used for offline analysis.
Defense Strategy	Deception strategy represents the global defense strategy for the overall system. It should conform to decoy policy and doctrine.
Policy / Doctrine	Policy and doctrine draw the framework in which the domain-based defense strategy is defined. Deception-based actions and tactics conform to this framework.
System Engineer	System engineers keep the decoy system operational. Although DB-IDRA is mostly automated, system engineers analyze the system performance and, if necessary, they update specifications for behavior patterns and decoy actions manually.

Table 1 Concept category list

C. CONCEPTUAL MODEL

The conceptual model in Figure V-2 (on the next page) represents the general concepts and their associations with each other. The model represents the functions to be performed by the system. The conceptual view in the diagram contains the concepts and their relations. The conceptual model does not show the architectural entities; however, it defines how the concepts and relations can be applied to the architecture.

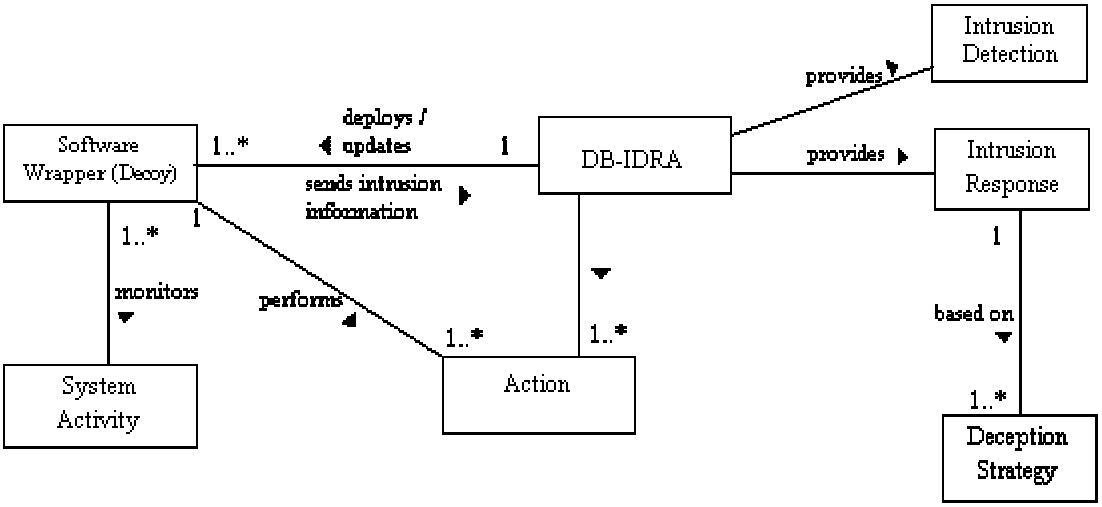

Figure V-1 Central functionality of DB-IDRA

FigureV-1 represents a partial view of the diagram concentrating on the central functionalities of DB-IDRA. The associations in the diagram are bi-directional, meaning that from each concept logical traversal to the other is possible. The little arrowheads indicate the direction of this logical traversal. Depending on the type of association, it is possible to have unidirectional associations. For example, the association between the *DB-IDRA* and *Intrusion detection* is interpreted as "*DB-IDRA provides intrusion detection*". On the other hand, the association between *DB-IDRA* and *Software Wrapper* is bi-directional, indicating data flow from each concept to the other. Software wrappers monitor the system activity and send intrusion-related activity reports to DB-IDRA. Then DB-IDRA sends back corresponding updates and deploys necessary actions. As a result, wrappers perform those actions. The diagram also shows that DB-IDRA provides intrusion response that is based on deception strategy.

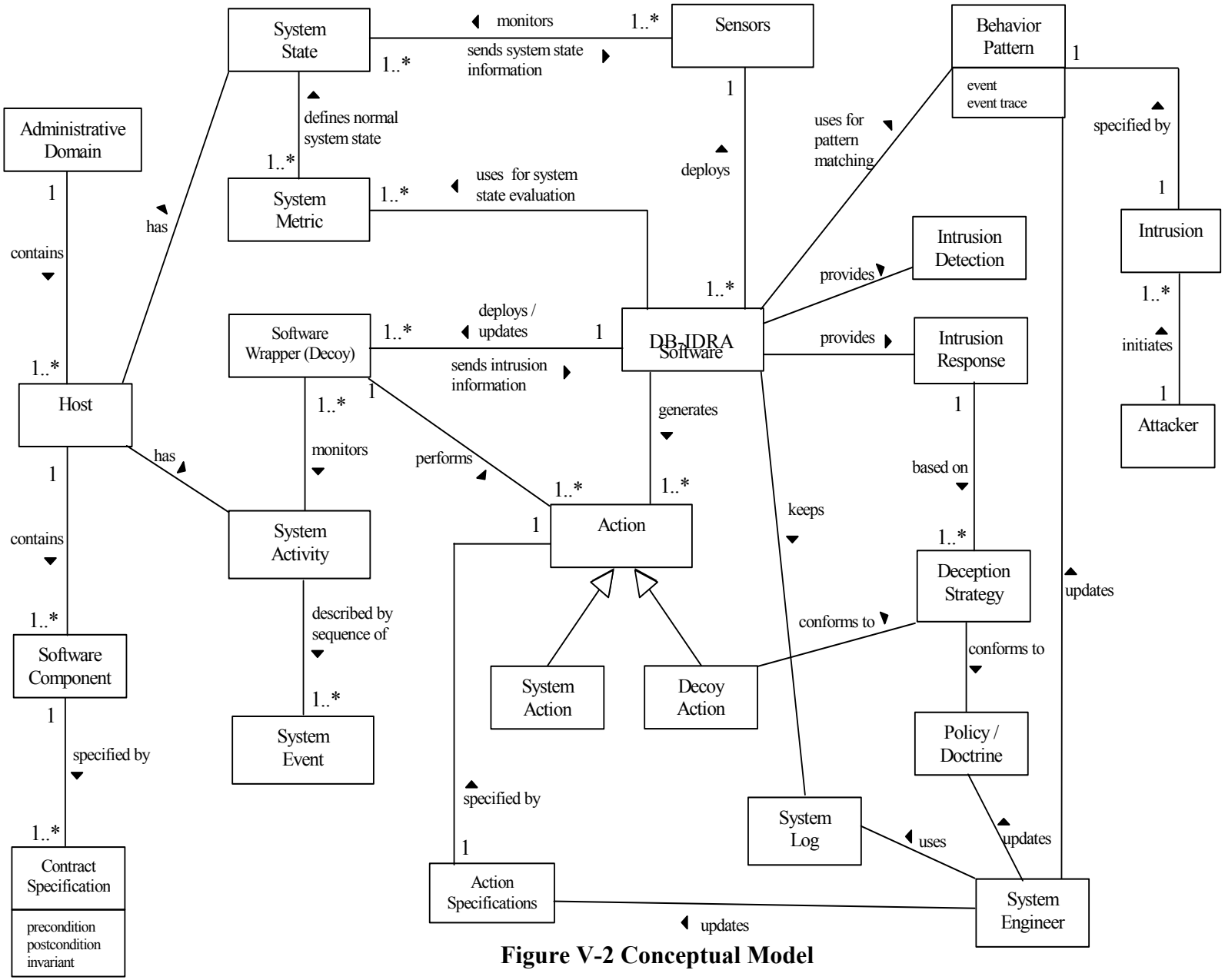

Figure V-2 Conceptual Model

After analyzing the concepts and the relations between them, it is possible to make preliminary decisions about the architecture. DB-IDRA should have at a minimum the following functionalities.

1. DB-IDRA should include a flexible intrusion detection mechanism that employs either anomaly-, misuse-, or signature-based detection technique.
2. The DB-IDRA should have an intrusion response mechanism that is based on deceptive defense strategy.
3. Kernel-based software wrappers (decoys) are used to detect and respond to attacks. Decoys should be reconfigurable and modifiable dynamically by a central decision-making module in order to handle runtime activities.
4. Deceptive tactics should be realized by decoy actions that are defined formally.
5. The defense-related activities should conform to a global decoy policy and doctrine.
6. While protecting host-based systems from cyber attacks, DB-IDRA should provide a defensive approach that must be scalable to a network and also should provide an interoperability framework among different administrative domains.
7. Besides intrusion detection and response, DB-IDRA should also take the system state into account in order to evaluate the impact of the defense activities on the system performance and QoS provided to legitimate users. Corrective actions (system actions) to optimize the system performance must be defined and integrated into the architecture.
8. In order to measure the system performance, a set of metrics should be defined for the system.
9. In order to detect intrusions and observe system state, DB-IDRA should employ an effective monitoring approach. Effectiveness, in this context, consists of timeliness, adaptability and robustness of the monitoring activities.
10. DB-IDRA requires a formal approach for specifying nominal system behavior, intrusion patterns, response actions and system actions.
11. DB-IDRA should provide a system audit log for offline analysis.

D. HIGH-LEVEL ARCHITECTURAL DESIGN

1. Overview

After requirements analysis and conceptual modeling, high-level architectural design is the first step towards a representation of the architecture in a form of software abstractions. Based on the functions defined in the previous section, high-level design involves describing the behavioral model for the DB-IDRA and then specifying the architecture in UML notation.

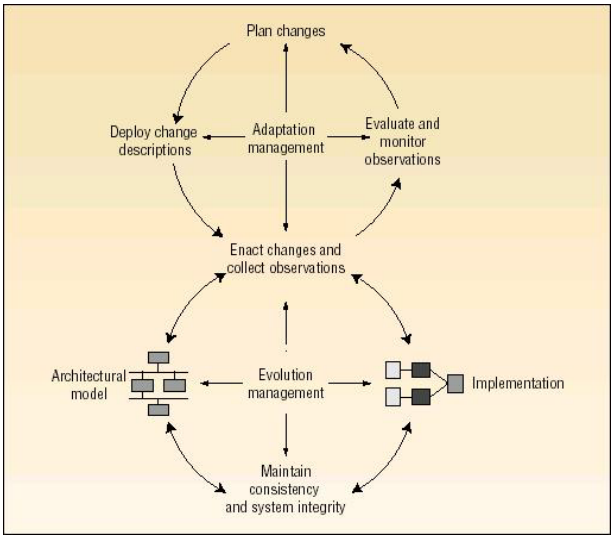

Figure V-3 High-level adaptation and evolution process (From: [36])

2. Dynamic Behavior Model

DB_IDRA includes five high-level functionalities:

- Monitoring,
- Intrusion detection,
- Tracking,
- Analysis, and
- Response.

These functionalities govern the runtime behavior of the system; therefore, the architecture needs to be designed considering how the system functions in a dynamic environment. Oriezy *et al*. [36] introduces an approach supporting two simultaneous processes to manage the dynamic structure of a system based on its software architecture: (i) e*volution*, the consistent application of change over time, and (ii) *adaptation*, the cycle of detecting changing circumstances and planning and deploying responsive modifications. In this approach, explicit consideration is given to the role of software architecture in planning, coordinating, monitoring, evaluating, and implementing

adaptive responses. Figure V-3 defines these two processes and their sub elements. The main distinction between the upper and lower portions of the diagram is that adaptation management describes the life cycle of adaptive software systems; on the other hand, the evolution management describes how adaptations are applied to adaptive software systems.

Although Oriezy *et al.* defines this approach for self-adaptive software architectures, it provides a generic framework in which dynamic behavior of a system can be modeled. In order to do that, we need to change our focal point to behavioral aspects of a system, rather than the architectural components. We envision that a dynamic adaptation and evolution process in our problem domain should address the following:

- *Monitoring.* Depending on the environment, the system should be able to update the granularity and the hierarchy of data collection and observation components.
- *Response.* As the intensity and characteristics of intrusions change, it is desirable to be able to update the defense strategy. This kind of update may be related to either low-level (e.g., deception tactics, atomic response actions) or global (e.g., dynamic policy update) decisions.
- *System state.* Protecting a system against attacks can have a negative impact on the system performance and resource usage. Therefore, observing and fine-tuning the system state, if necessary, against these fluctuations is important.

Figure V-4 shows the modified version of the process defined in Figure V-3 for DB-IDRA. Adaptation management describes the runtime decision-making process, while evolution management focuses on the mechanisms employed to change the runtime configuration of the system. DB-IDRA represents a dynamic behavior; that is, it must update its behavior at runtime according to the current system state. To do so, it must collect data, analyze the data, and deploy necessary responses based on analysis results. The process characterizing the behavior model must, at a minimum, manage those three basic functionalities comprising *adaptation management*, and maintain consistency and integrity of the system via *evolution management*

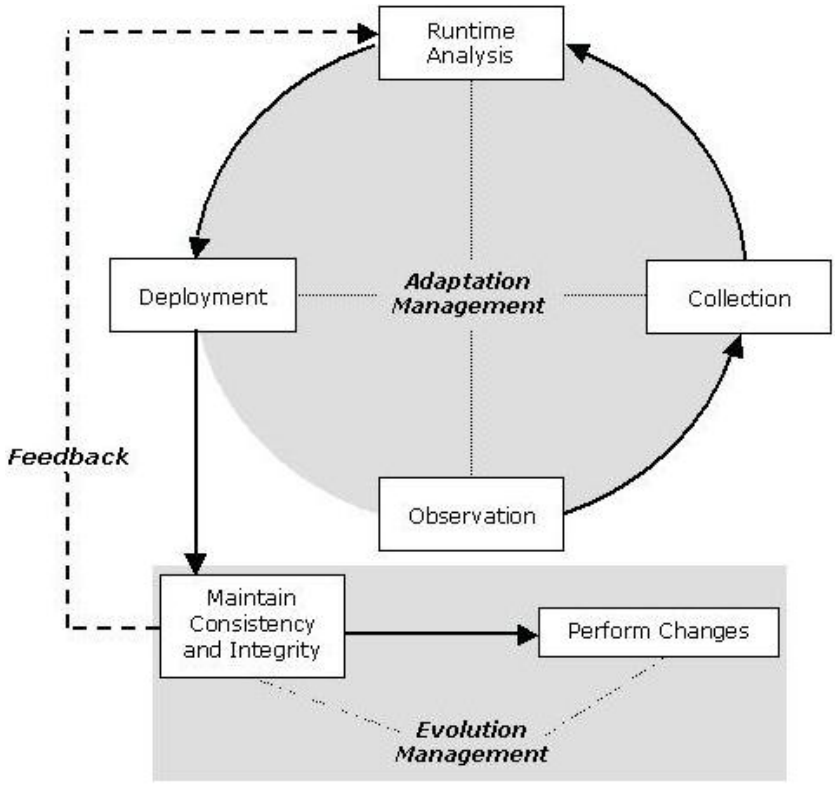

Figure V-4 Process diagram

3. Adaptation Management

Adaptation management describes the process from data collection to change specification deployment. The adaptation process defines the activities from when an observation is reported, until necessary actions are deployed, as illustrated in Figure V-5.

Observation is the initial step in adaptation management, which gathers raw data about individual components of the system via *observers* and passes those data to collectors. These data provide information about both intrusive processes and system state. Intrusion-related data are used to determine if there is an intrusive interaction with the system. Data about the system state indicate the necessary system properties to evaluate the environment regarding performance, efficiency, stability and so on. Observers may be expanded with the filtering functionality to decide whether to keep a record of an observed event or to ignore it.

The *Collection* step involves gathering raw data from observers and repositories with the data. It is possible to place several collectors in a system, providing an opportunity to monitor the system from different views at each collector. Communication

between the observers and the collectors can be established via either sending data on a shared medium (advertising), or using queries to get the data individually (soliciting) [35]. Collectors also store a *system log* containing past data about the system activity. This log can be used for offline analysis to evaluate how well it operates in terms of correctness, performance, efficiency, and so on. The number and size of buffers (required for this log) can be adjusted to store only the essential data for the defensive-related activities.

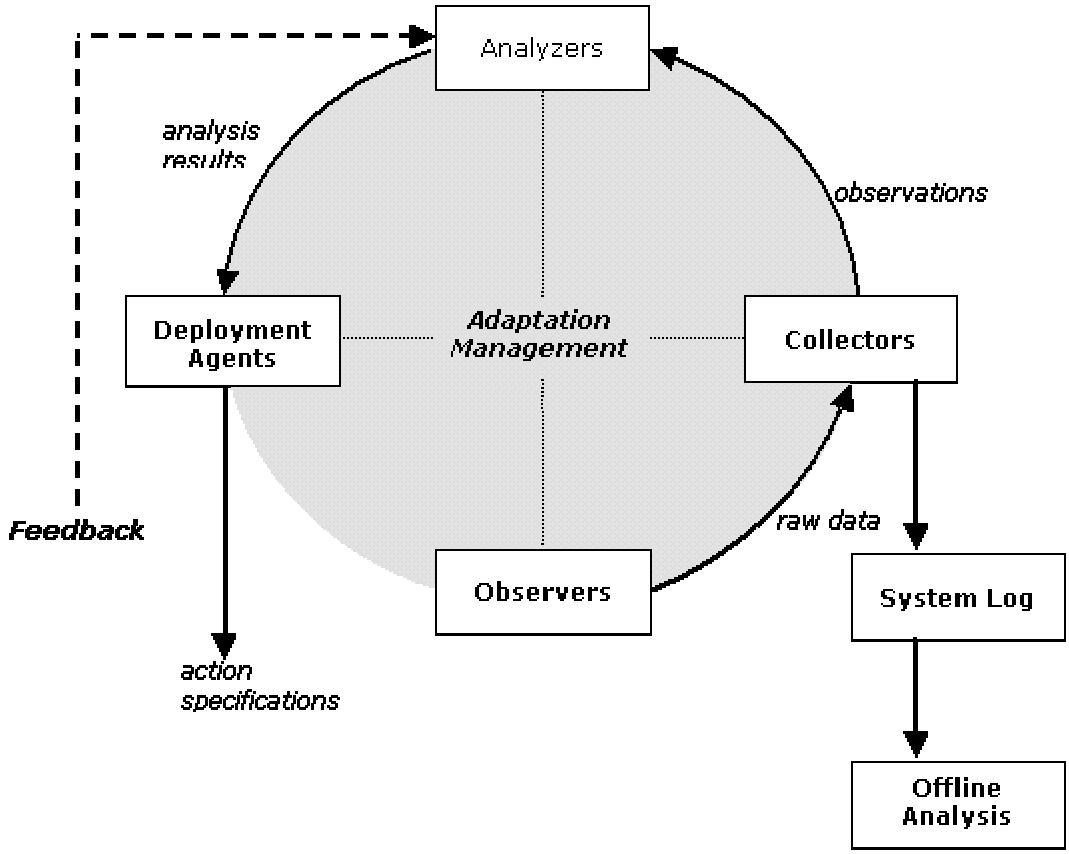

Figure V-5 Adaptation management process

Runtime analysis refers to the analysis of the observations passed by the collectors. DB-IDRA performs analysis for intrusion detection and system optimization. For this purpose, the analysis package employs different analyzers for each task, which are called *intrusion* and *system analyzers*. Based on the intrusion detection technique used in a system, intrusion analyzers work with a corresponding database storing the detection rules. This approach provides a flexible framework in which any intrusion detection technique (e.g., anomaly, misuse, and signature-based) can be used independently from the rest of the system. On the other hand, system analyzers make

decisions about the system state by analyzing some prespecified system variables. That repository contains the formal specifications for behavior patterns for intrusions and response actions. Observations are compared to those behavior patterns, and then necessary actions are produced as a result. It is also possible to delegate some of the primitive analysis requirements to observers, such as counting and time stamping.

Deployment is the last step in adaptation management. The deployment module employs agents to carry out the specified actions. At this point, action specifications are distributed among agents in an organized way. Actions' being coordinated or not has an impact on this distribution process. Deployment agents interact with the supervisor. Once the actions associated with agents are approved, agents proceed and modify the wrappers according to the transactions contained within the action specifications.

4. Evolution Management

It is not enough to decide what to change in a system at runtime. There are several challenges with respect to how those changes are integrated to the running configuration of the system. These challenges may be related to safety, reliability, consistency, integrity and correctness. There are two possible scenarios that might cause an unintended outcome. In the first scenario, the adaptation process may decide on an ill-considered action. In the second scenario, the decision made by the adaptation process is added to the system incorrectly. For example, an intrusion detection mechanism may decide on blocking the *http* channel; however, this might ruin an ongoing deception tactic for interacting with an attacker.

In order to maintain consistency and integrity of the system, evolution management defines the approach in which actions are incorporated into the system, as shown in Figure V-6. Actions produced by adaptation management are reflected in the system's current state, while ensuring that those actions are consistent with the global security policy, the current defense implementation, and the system configuration. The results of the activities in the supervisor are fed back to the adaptation management level mechanisms. The evolution management introduces two types of components to accomplish its tasks: *supervisor* and *wrappers.* The supervisor verifies that the actions do not conflict with the consistency and integrity of the system. In this context, consistency implies that the actions should conform to the global defense policies. Furthermore,

integrity ensures that all the defensive actions accomplish the same goal without conflicting with each other. Wrappers are the kernel-based modules that can respond to intrusive processes.

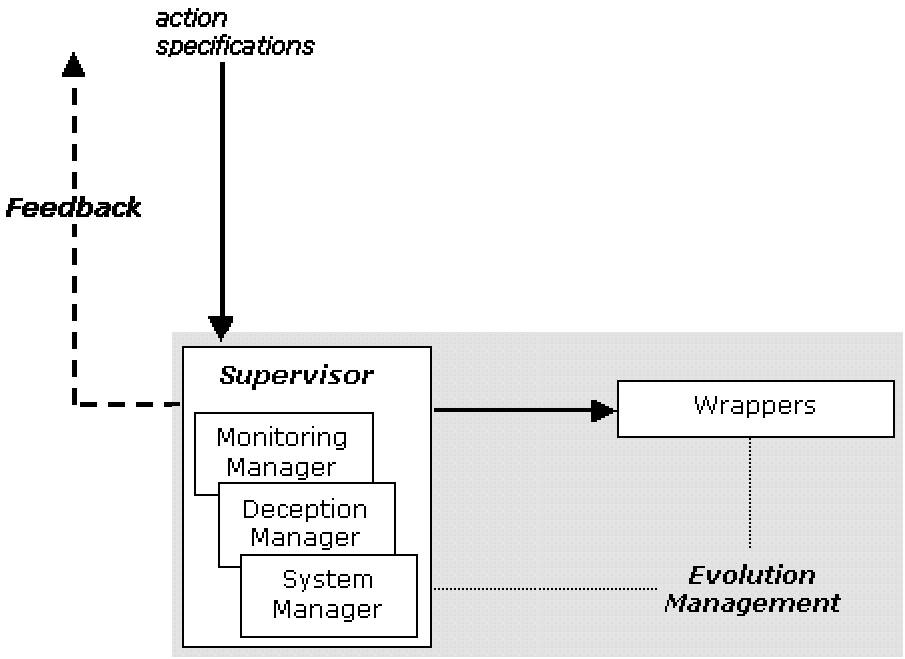

Figure V-6 Evolution management process

The supervisor contains three sub components to manage the system consistency and integrity from three different perspectives. The *Monitoring Manager* analyzes the actions that change the monitoring configuration of the system. The *Deception Manager* analyzes the deceptive actions against intrusions. The *System Manager* analyzes the actions that change the system configuration for optimization purposes. These components depend on each other and the global state of the system. For example, as the intensity of attacks against a system escalates, resources used for maintaining the global defense strategy also increase. In that case, the system may not be able to accommodate more deceptive actions and may need to adjust some of the runtime variables to keep the system stable. The system should be able to continue servicing its legitimate users, and DB-IDRA should still be able to protect the system, even with degraded capabilities. Any action conflicting with the criteria set by the supervisor is vetoed and the necessary feedback is sent back to the adaptation layer. Approved actions are sent to the wrappers and subsequently activated.

Wrappers are kernel-based modules that perform the intrusion detection and response actions. Within the scope of the evolution layer, wrappers respond to intrusive processes with the actions passed by the supervisor. Alternatively, wrappers can be used for intrusion detection; in fact, detection and response functionalities can be combined in wrappers. In this regard, observers mentioned in the adaptation layer can be implemented as wrappers too.

5. Architecture Package Diagram

To illustrate the system architecture, we used the UML *package* construct. Packages contain groups of elements or sub systems such as components, use cases, or other packages. Representing the architecture in a multi-layered approach provides several advantages. First of all, it isolates application logic into separate components that can be reused. Thus, it allows designers to organize their ideas about how to distribute the tiers on different physical computing entities. Figure V-7 shows the architecture package diagram for DB-IDRA. This diagram includes the logical groupings that will be expanded in the detailed architecture presented in the next section.

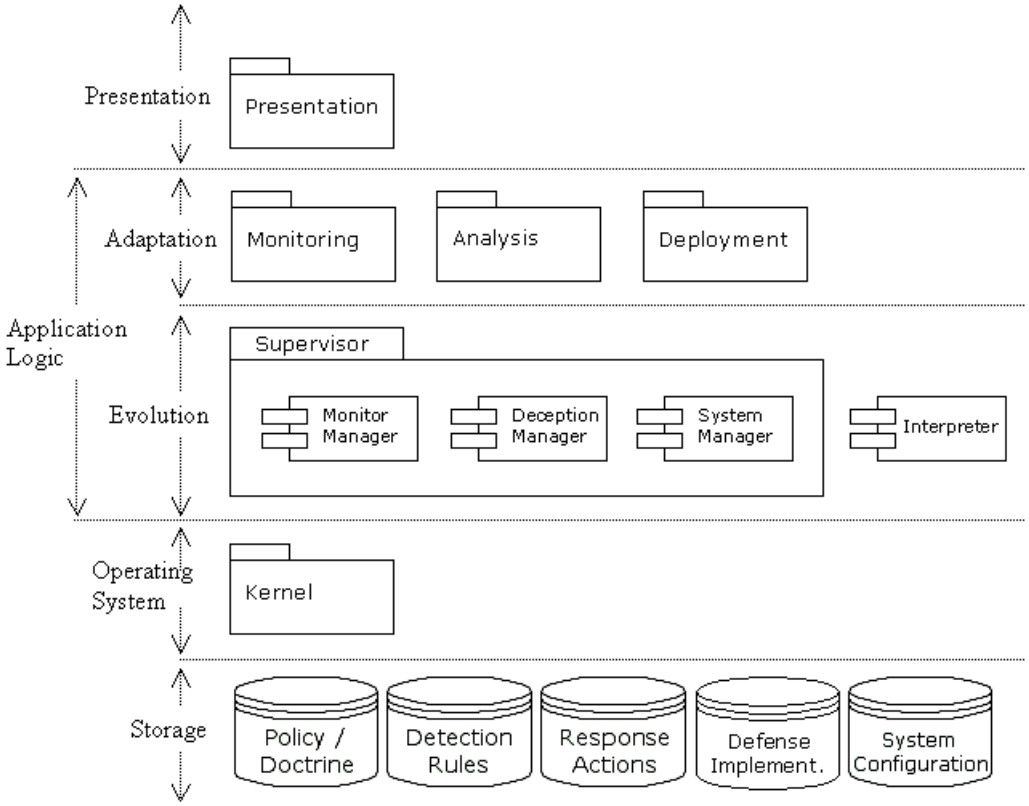

Figure V-7 Architecture package diagram

DB-IDRA is separated into five layers: *presentation, adaptation, evolution, operating system,* and *storage.* Although presentation functionality is important, we think it is sufficient to note that the system should provide a user interface. Our main focal point is the underlying architecture.

Application logic consists of the *adaptation* and *evolution* layers. These two layers describe the tasks and rules that govern the entire system behavior. Following the previous section on behavior model, there is a continuous interaction between the adaptation and the evolution layers. The o*perating system* layer consists of the kernel-based modules to perform response actions and monitoring activities (if monitoring is done in the kernel space). The s*torage* layer provides the persistent storage mechanism that serves both the adaptation- and evolution- layer components.

VI. DETAILED ARCHITECTURAL DESIGN

A. OVERVIEW

In the previous chapters, we explored the problem domain and articulated an initial set of requirements for the system, from which we then derived both a conceptual model and high-level architecture. Our design reflects a four-layered architecture: *presentation, application-logic, operating system* and *storage* layers. Each layer represents a modular entity containing sub components to perform the entity's assigned tasks. The Detailed Architectural Design phase further defines these packages, their sub components, and the intra- and inter-package collaborations.

B. DETAILED ARCHITECTURAL DESIGN

DB-IDRA is designed in a layered approach. The adaptation and evolution layers (these two are sub layers of the application logic layer) constitute the underlying logic of the system. The operating system layer organizes the software wrappers for detection and response purposes. The storage layer provides the common shared database structure, whereas the presentation layer allows system engineers to interact with the system both at runtime and offline. Each layer represents a logical grouping assigned a specific set of tasks. For example, the monitoring package consists of observers and collectors to gather data about the system and perform primitive actions (e.g., simple preprocessing of the sensor data). The layered architecture has interfaces between the layers and their sub elements. The following sections describe each package in detail and how the packages interact with each other.

C. PACKAGE DESCRIPTIONS

Packages are general-purpose mechanisms for organizing elements into groups. Packages are purely logical unlike components. They only exist at development time, for the most part, to manage complexity. This section describes each package included in the architecture package diagram. Package descriptions address the static structure of the architecture and address key aspects of the dynamic behavior of the system.

1. Presentation

The presentation layer encompasses the user interface, which is to be used to monitor system behavior and modify the system configuration in order to coordinate the system's behavior. The Presentation layer can be grouped into *runtime* and *offline* functions. In each category there are some key design considerations that are generally applicable to any kind of user interface. Since intrusion detection systems perform time-critical tasks, the user interface should be capable of presenting information about the entire system in a bounded response time. Additionally, it should have notification mechanisms to let the system engineers know when prespecified conditions occur. Besides timeliness, information organization and level of detail must be considered in the presentation layer design too.

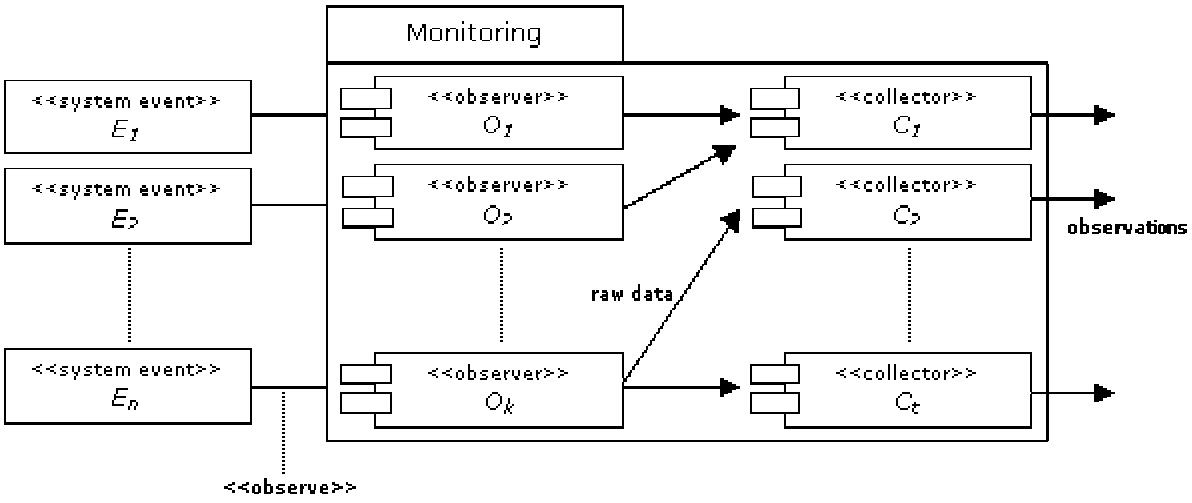

Figure VI-1 Monitoring Package

2. Monitoring

The Monitoring package is composed of observers and collectors. Observers are the lowest level entities interacting with the system components, and sense system-level events. Collectors aggregate and perform postprocessing on sensor data. Observers and collectors can be organized in a hierarchical monitoring structure. Each collector also checks that the observers are operating normally. If a collector detects a malfunction in any of its observers, then the collector alerts the analysis package about the malfunction.

Figure VI-1 illustrates an example of how monitoring components can be organized. The observers collect data about different aspects of the system activities. System activities are composed of events (E_1, E_2...En). In the diagram, E_1 can be events related to all or a specific set of http requests and replies. The observers (O_1, O_2...O_k) can either collect all data on system events or selectively collect data by using prespecified rules as a guide. As independent entities, observers can be turned off and on during runtime. On the other hand, collectors provide different views over the same set of observers. For example, C_1 gathers data from both O_1 and O_2, where C_2 and C_t collect data from O_k only. This kind of an organization provides for flexible monitoring capabilities. For example, different groupings of collectors (like C_1 and C_2) can focus on different aspects of the system (e.g., events related to http, ftp, and DNS). Additionally, collectors can share the same observers (e.g., C_2 and C_t) but concentrate on different properties (e.g., attributes) of the events being passed by those observers.

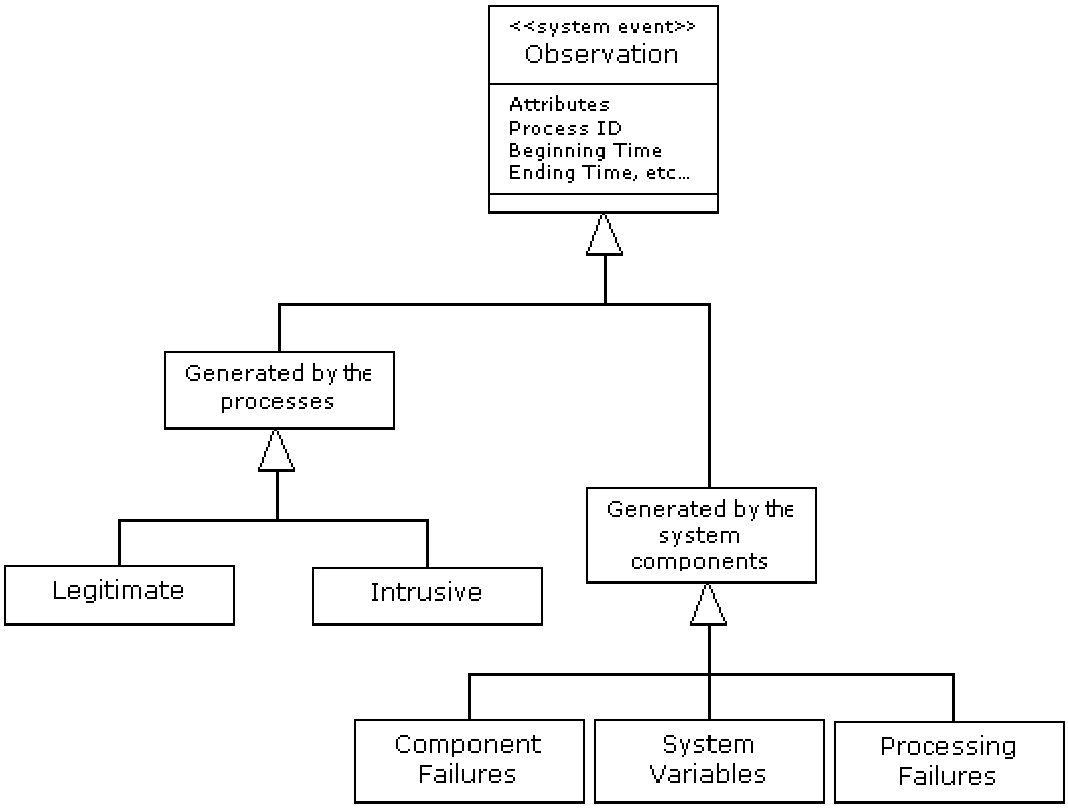

Figure VI-2 Observation categories

The monitoring package passes the observations to the analysis package. First, observers gather the data and filter it if necessary, and then collectors group and pass those data to the analysis package.

Observations are the basic system events, and they are grouped into two main categories, as shown in Figure VI-2. The first category includes the events generated by the processes running in the system. This category consists of legitimate and intrusive events. The second category includes the events generated by the system components to indicate a specific situation. Events generated by the system components include component failures, system variables, or processing failures.

3. Analysis

The Analysis package provides the functionality to analyze observations from three different perspectives: *intrusion detection, response,* and *system optimization.* To do that, the analysis package employs analyzers. Currently, we envision three types of analyzers: *intrusion detector, system optimizer,* and *response generator*. However, new analyzers can be added to the package or the existing ones can be modified. The analysis package employs a tracking component to isolate intrusive processes from the nominal system mode. This isolation enables the system to apply deception to keep the intruders uninformed about the actual system state, while gathering information about the nature of these processes. Figure VI-3 shows the internal structure of the analysis package.

The *intrusion detector* performs intrusion detection based on the detection techniques employed in the system, which could also be hybrid (e.g., anomaly and signature-based detection). The detector analyzes the observations according to the database of detection rules. The intrusion detector interacts with the tracking component to compare the current behavior of an intrusive process to that process's past behavior. If a process is considered to be intrusive, the specifications about the intrusion are passed to the response generator.

The *system optimizer* obtains system state and configuration. The optimizer gets that kind of data via *probes*. Probes are specific-purpose observers to gather data about system variables. Data received from probes are checked against predefined limits for the nominal system operation. Those limits are defined by metrics, such as performance, stability, and efficiency. For example, the system resources are shared by the legitimate services and intrusion detection activities. It would not be logical to dedicate all the system resources to the security activities at the expense of all the other system services. Therefore, optimizing the resource usage is necessary for an efficient runtime performance of both the security and all other legitimate activities. Based on the analysis performed by the optimizer, specifications about system actions are passed to the response generator.

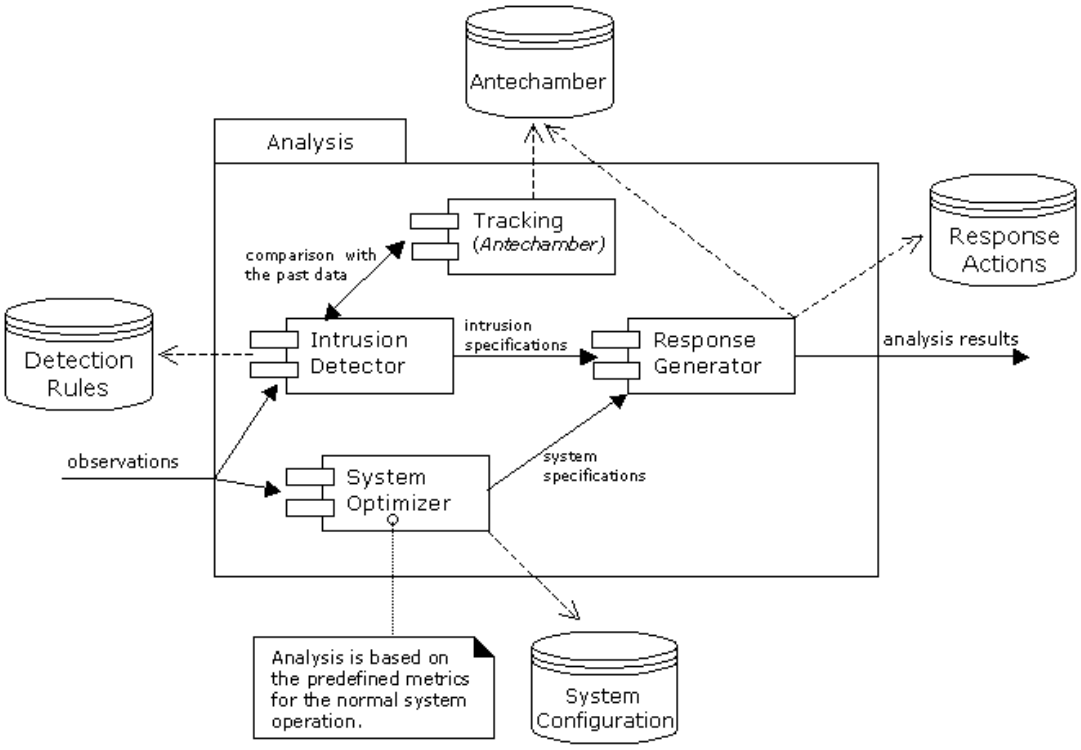

Figure VI-3 Analysis package

The *response generator* takes the intrusion and system specifications as inputs and produces the analysis results containing the corresponding action specifications. Those specifications are based on the action rules stored in the database. The response generator takes into consideration not only the response rules but also the current defense implementation.

The *tracking component* isolates intrusive processes from the normal system context and keeps track of their behavior and the responses against them. All the interactions with the intrusive processes are managed by the tracking component and the data about those interactions are stored in a database. This component provides an interface to the other components requesting services about the current status of the defense implementation.

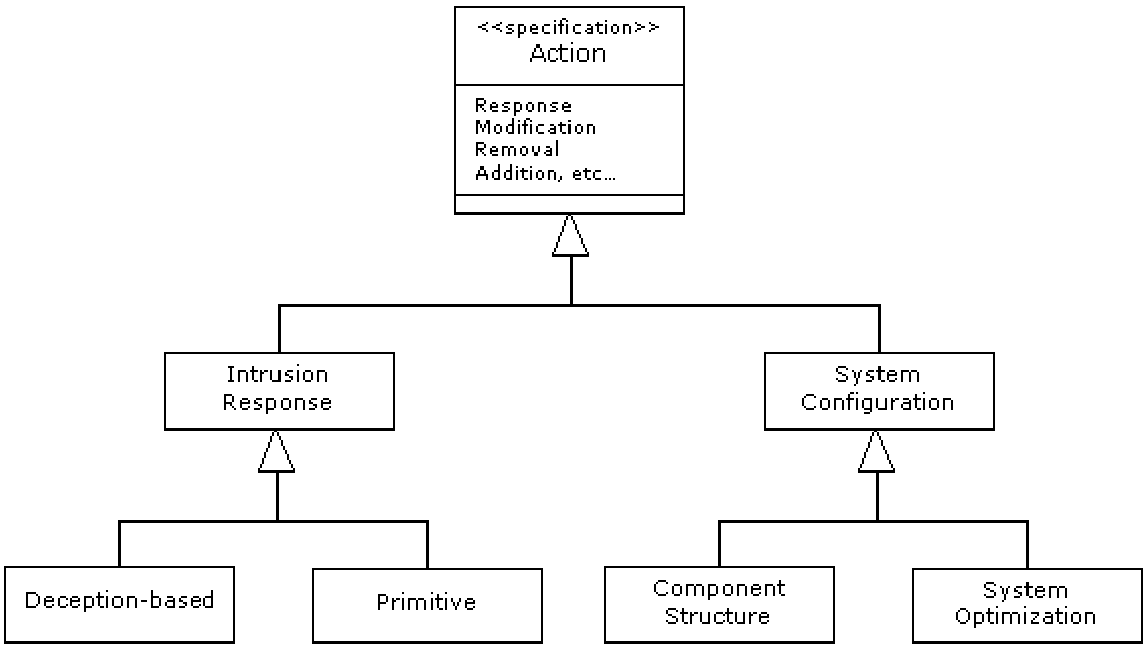

Figure VI-4 Categories of actions generated by the analysis package

The analysis package passes the analysis results to the deployment package. Analysis results are defined in two types of actions, as shown in Figure VI-4, changing different aspects of the system: *intrusion response* and *system configuration actions*. Intrusion responses can be either deception-based or primitive actions. System configuration actions either change the components' structure or optimize system performance.

4. Deployment

When the response analyzers pass the analysis results, the *distributor* assigns the responses to *deployment agents*. The basic criterion for this is whether the actions are coordinated. Therefore, each response specification defines an agent's task. However, agents can collaborate with each other to realize coordinated responses. $Agent_1$ and $Agent_n$, in Figure VI-5, can participate in a coordinated response as they propagate

through the system and carry out their mission individually. In case of coordinated attacks, deceptive responses are deployed separately for each process playing a part in that attack. However, the necessity of coordination is obvious considering the need to present a consistent deceptive counter measures to the attacker. In addition, agents can perform responses against attacks, for instance, directing an intrusive process to a fake resource.

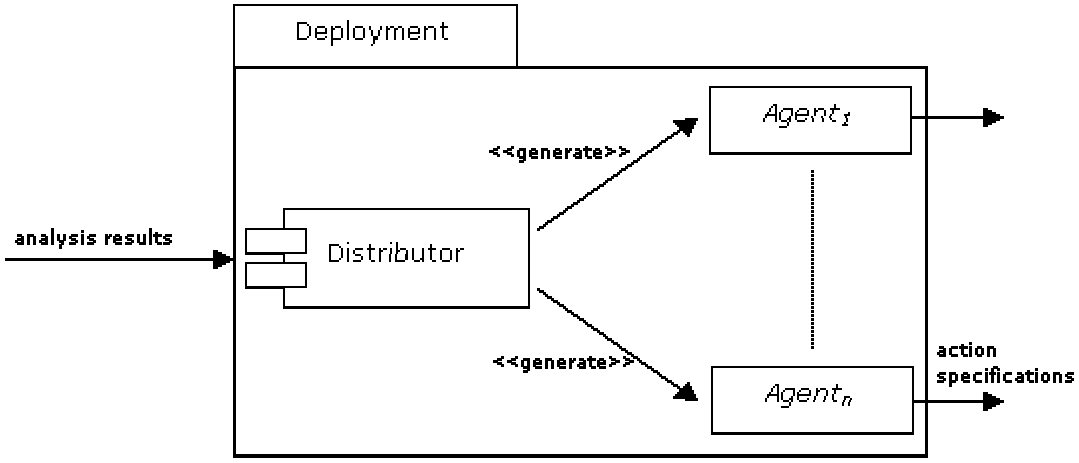

Figure VI-5 Deployment agents

5. Supervisor

The Supervisor is the main evolution layer component containing three sub components: *monitor, deception,* and *system managers*. The Supervisor is responsible for maintaining the consistency between the global security policy and the current defense implementation. One of the essential properties of the supervisor is that it has the right to approve or veto any change specification passed by the adaptation layer. Modification of the inappropriate changes is not part of the supervisor's responsibilities.

Figure VI-6 shows the interdependencies between the components of the supervisor. For example, if the intensity of the intrusions is high and the system is not able to allocate any more resources for intrusion detection and response activities, it may be required to rely on less computationally-intensive deceptions or reduce the number of observers in order to maintain acceptable level of QoS.

Monitor planning determines which observation components are necessary for monitoring the system activities. Planning takes into account the system performance and the intrusion load on the system. The monitor planner can reconfigure the structure of the monitoring components dynamically. *Deception manager* examines the changes before they are applied to the current system implementation. The deception-based responses must conform to both the global security policy rules and the current defense implementation. The s*ystem manager* determines how the changes will affect the system performance. Optimization is necessary for maintaining the legitimate service of the system, while protecting it from cyber attacks. Each component uses its veto option if an action conflicts with the criteria set by the supervisor and sends feedback to the adaptation layer mechanisms.

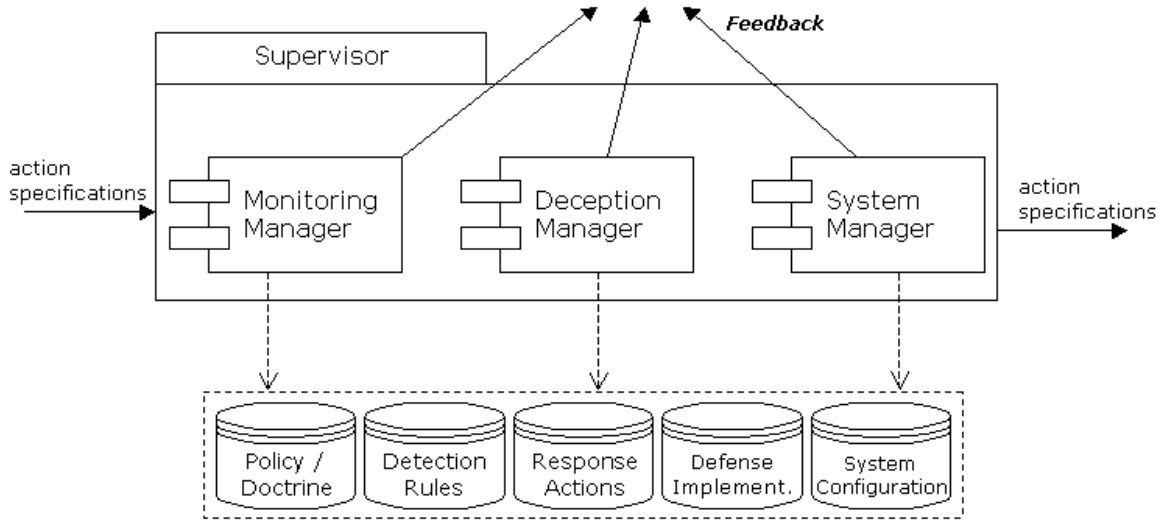

Figure VI-6 Internal structure of the supervisor

6. Interpreter

The interpreter is a compile-time component that translates high-level specifications for deceptions into low-level wrapper language. This component is used in preprogramming the kernel-resident modules. As a result of this compilation, wrappers are equipped with both detection and response capabilities.

7. Operating System

The Operating System package includes the kernel-based intrusion detection and response modules (i.e., wrappers). These modules can perform intrusion detection, response tasks, or both. Observers, mentioned in the monitoring package, can actually be

implemented as kernel modules too. Wrappers are preprogrammed with detection and response capabilities at compilation time. Then these capabilities are configured at runtime as the system context changes. The essence of the kernel-based approach is that complicated tasks should not be carried out in the kernel. Wrappers provide the front-end of the security measures against the attackers, while benefiting from the additional secure environment of the kernel space.

8. Database

All the packages and components perform their tasks based on rules, specifications or configurations. There are at least five groups of data shared by the components: *policy/doctrine, detection rules, response actions, defense implementation,* and *system configuration.*

- The *policy/doctrine* database draws the framework in which the domain-based defense strategy is defined. The supervisor maintains the consistency and integrity between this framework and the current defense implementation. These rules represent legal, strategic and administrative issues that might affect the course of actions against cyber opponents. Obviously, the policy and doctrine for different administrative domains, such as military and nongovernmental organizations, dictate different rules according to their legal status, assets being protected, defense approach, and so on.
- The *detection rules* define the intrusive behavior patterns for the detection technique being used in the architecture. This database is used for two different purposes. In the first case, the wrappers are initially equipped according to the rules stored in this database. In the second case, the intrusion detector uses these rules to detect intrusions at runtime.
- The *response actions* define both deceptive and primitive actions against attacks. This database represents a wide spectrum of actions to increase the efficiency of responses. The response generator selects response actions based on the input from the intrusion and system analyzers.
- The *defense implementation* database contains the data about the ongoing intrusion-related activities in the system such as the current intrusive processes, behavior history of and past responses against those processes. This database affects intrusion detection, response generation and supervisor activities. This database provides the capability to keep track of these isolated intrusive processes. If the policy/doctrine database is the generic model for the system, then the defense implementation is the realization of the model. The defense implementation database is a dynamic data structure, which means, it is updated as: New intrusions (or suspicious behaviors) are detected; processes perform new actions,

interactions with the processes are terminated; and processes are considered to be legitimate and released from the antechamber.

D. NON-FUNCTIONAL REQUIREMENTS

Software architectures are derived from functional and non-functional requirements. The purpose of having two types of requirements is to reduce the complexity of the design by separating what the system does from how the system behaves with respect to some observable attributes such as performance, reliability, and stability. While functional requirements can easily be observed in an architecture supported by an inclusive documentation, that is not the case for non-functional requirements. Although, the considerations regarding this group of requirements are specified in advance, it is important to verify that the architectural design complies with the non-functional requirements.

We have introduced six levels of non-functional requirements for the architecture: *interoperability, stability, survivability, scalability, efficiency,* and *performance.* We chose to focus on the non-functional qualities preserving distinct characteristics for the design, rather than the common ones such as reliability and maintainability. We believe that it is possible to realize this architecture in an implementation meeting all of these qualities, but this remains to be demonstrated through a detailed case study.

Interoperability represents the need for effective communication and collaboration means between different administrative domains. As an administrative domain collects valuable information about attacks, sharing that information with other domains allows for establishing proactive protection measures. That kind of information can be directly inserted into the databases for the analysis mechanisms to use. For example, if a domain has detected a new type of attack, then it shares this information with other domains, thereby permitting the other domains to take precautions to detect and perform deceptions in response to that attack. As the number of collaborating domains increases, interpretation of data from multiple sources becomes more complex. With its current design, DB-IDRA can use the information provided by external sources. This information can be either inserted into the databases or fed into the runtime process as system events, provided that a common communication language is already developed among collaborating domains.

Stability of the runtime architecture necessitates that the system should confront as many attacks as possible to keep the overall system optimized. It is also necessary to evaluate the stability under different levels of intensity of conflicts to make sure the system operates as intended. We can measure the level of stability based on the network load. The network load can be separated into two types of interactions: *legitimate* and *intrusive processes*. The system can adjust itself by maintaining a threshold ratio between these two groups of users at any point in time. For example, if a system allows only one third of the system resources to be used for intrusive processes, then the intrusion detection mechanism adjusts the current defense implementation based on this threshold. Besides network load, the complexity of ongoing deception activities can consume system resources if they are at a sophisticated level such as mimicking the operating system. Network load and deception activities are concerned with effective resource allocation among legitimate and intrusive processes; however, failure of the system components can also cause the system to transition into unstable states. The supervisor needs to mitigate the effects of such disturbances.

Survivability, in its basic definition, is a system's capability to fulfill its mission (as desired) in the presence of attacks, failures, or accidents. The survivability process represents different characteristics for every system depending on the essential services and assets contained in it [38]. Our architecture provides a deception-based intrusion detection and response framework. It may be possible, for instance, to use redundancy as a mean to increase the survivability of the system.

Wrappers are used in the kernel space. Wrappers modified for different purposes including intrusion detection and response. They provide an autonomous front-end against attacks with the benefit of additional kernel protection mechanisms. In case of a failure within one of the supervisor components, the system can still provide intrusion detection services. This can be accomplished with the primitive logic preprogrammed into the wrappers. For example, if the deception manager fails, DB-IDRA can deactivate deception functionality, but still provides intrusion detection without deception.

There are several possible ways to distribute supervisor capabilities throughout a network in order to achieve redundancy. The network can have its own supervisor with

full functionality as defined in earlier sections. If any of the hosts fails, then another host takes over the role of the failed host. Another approach is to organize supervisors in a hierarchical manner. Figure VI-7 illustrates that kind of a scenario for a network of eight hosts. In that case, each host can have either a fully functional supervisor, a supervisor with limited functionality or no supervisor at all. This approach supports the implementation of inter-domain level survivability strategies.

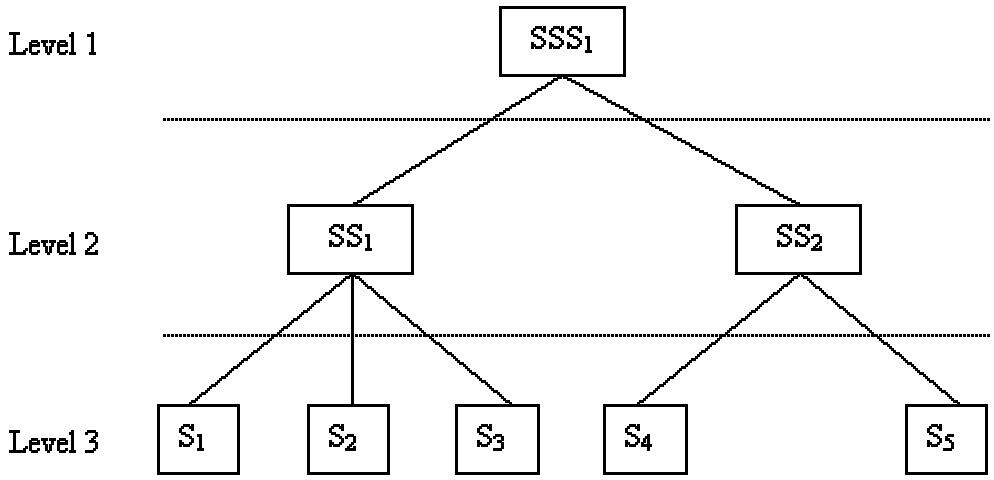

Figure VI-7 Hierarchical supervisor organization

DB-IDRA represents a *scalable* architecture. The architecture describes a host-based system; however, there are some distinct properties that enable the architecture to be scaled up for use in large distributed systems. Wrappers, in the architecture, include self-controlling primitive logic mechanisms. This reduces the data transferred to the upper layers, because unnecessary data is filtered out at the kernel layer. While each host can manage its defense and decision making activities by itself, network-wide analysis and defense need to be managed by a higher-level entity. In our architecture, this can be done via hierarchical organization of supervisors, as described in Figure VI-7. At level 3, each host manages its own protection. Level 2 represents sub domains in the network; in that case, SS_1 and SS_2 are responsible from three and two hosts, respectively. These sub domains can be established along with physical boundaries, such as subnets, or some other logic that might promote the robustness of the system. The information flow from level 3 to level 2 includes only the essential data that contributes to a sub domain's coordinated activities. Level 1 supervisor (SSS_1) collects data from sub domain supervisors (SS_1 and SS_2) to make further decisions about the network. This highest-level

decision making unit is responsible for the entire domain, so it also communicates with other domains to support interoperability.

Performance is a general constraint over all the functional and non-functional requirements so that necessary means must be provided to minimize the affect of the monitoring and decoy actions (*e.g.*, those of delay tactics) on the availability and performance of computing resources requested by legitimate users of the protected software components. DB-IDRA observes the runtime system performance and dynamically adjusts system variables to keep the system within predefined limits like resource usage, network load, and so on.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CONCLUSION AND FUTURE WORK

A. SUMMARY

The scope of this thesis study was to design a generic software architecture for deception-based intrusion detection and response systems. We demonstrated a systematic process for architecting deception-based intrusion detection and response systems. We followed a disciplined approach starting from requirements analysis to detailed architectural design. UML notations provided for representing the architectural abstractions.

We started by analyzing the requirements for the decoy-based system. Since the intelligent software decoy concept is still evolving, we spent most of our efforts to extract essential requirements from the theoretical and practical results presented in related publications. We incorporated a set of non-functional requirements into the design activities, which drove the design decision throughout the development process.

Based on these essential requirements, we developed the architecture in three consequent phases. In the first phase (i.e., Plan and Elaborate), we explored the domain and produced the requirements and the use cases. The second phase (i.e., High-level System Definition) used the artifacts produced in the initial phase to come up with the high-level definition of the architecture in a form of software-related abstractions. At this level, the architecture is organized in a layered structure, each layer representing a logical grouping of functional properties allocated to architectural packages and components. The last phase (i.e., Detailed Architectural Design) further defined these.

The end product of this study is the software architecture for deception-based intrusion detection and response systems. The resulting architecture is generic enough to be flexible. For example, the architecture does not specify a particular intrusion detection technique. It includes the constructs that are necessary to implement the intrusion detection mechanism regardless of the technique being used. A major contribution of the research is a proposed way to incorporate deception into software architecture. The architecture addresses production, application, and management of deception-based activities. However, the details about theory of deception are out of the scope of this

study. The architecture also serves as a guide for optimizing system performance at runtime to mitigate the performance overhead introduced by the defense activities. Lastly, the UML diagrams provide unambiguous representations of the system architecture.

B. FUTURE WORK

A logical follow-up to this study would be to complete the development process and implement the architecture, resulting in a prototype. As the concept evolves, the prototype would also be updated. The value of prototyping in software development is especially significant in demonstrating evolving design concepts and facilitating communication between business customers and technical developers. Instead of implementing the whole architecture, a logical subset can also be implemented such as response generation, system monitoring, and system optimization.

However, formal definition and representation of the architecture presented in this thesis may provide additional benefits before the implementation. For formal definition of the architecture, architecture description languages (ADL) can be used [28] and [29]. ADLs are used to define and model system architecture prior to system implementation. Further, ADLs typically address much more than system structure. In addition to identifying the components and connectors of a system, ADLs typically address: (i) Component behavioral specification, (ii) Component protocol specification, and (iii) Connector specification. A detailed analysis of the existing ADLs is required to decide which one is appropriated to use for this purpose.

A central aspect of architectural design is the use of patterns and architectural styles. Intrusion detection systems have commonalities in their functionalities and architectures. For example, the architecture represented in this study can be implemented on top of anomaly, misuse, or signature-based intrusion detection techniques. As long as the interface between the detection and response mechanisms is defined formally, the only difference would be related to the characteristics for each kind of detection technique. Therefore, it is logical to conclude that an architectural style can be developed for the systems that have deception-based intrusion detection and response capabilities. An architectural style has a number of practical benefits [41].

- It promotes design reuse
- It can lead to significant code reuse
- It is easier for others to understand a system's organization if conventionalized structures are used.
- Use of standardized styles supports interoperability.
- By constraining the design space, an architectural style often permits specialized, style-specific analyses.
- It is usually possible to provide style-specific visualizations.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Michael, J. B., and Riehle, R. D., "Intelligent Software Decoys," *in Proc. Monterey Workshop: Eng. Automation for Software Intensive Syst. Integration*, Monterey, California, Naval Postgraduate School, pp. 178-187, June 2001.

[2] Michael, J. B., Auguston, M., Rowe, N. C., and Riehle, R. D., "Software Decoys: Intrusion Detection and Countermeasures," *in Proc. IEEE Workshop on Information Assurance*, New York, West Point, pp. 130-138, June 2002.

[3] Michael, J. B., Fragkos, G., Auguston, M., "An Experiment in Software Decoy Design: Intrusion Detection and Countermeasures via System Call Instrumentation", *in Proc. IFIP Eighteenth Intl. Information Security Conference*, Kluwer Acad. Publishers, Athens, Greece, May 2003.

[4] Rowe, N.C., Michael, J. B., Auguston, M., Riehle, R. D., "Software Decoys for Software Counterintelligence," *IANewsletter*, v. 5, no. 1, pp. 10-12, Spring 2002.

[5] Michael, J. B., Wingfield, T.C., "Lawful Cyber Decoy Policy", *in Proc. IFIP 18th Intl. Information Security Conference*, Kluwer Acad. Publishers, Athens, Greece, May 2003.

[6] Michael, J. B., "On the Response Policy of Software Decoys: Conducting Software-based Deception in the Cyber Battlespace," *in Proc. the 26th Annual Computer Software and Applications Conference*, IEEE, Oxford, England, August 2002.

[7] Rowe, N.C., "Counter planning For Multi-Agent Plans Using Stochastic Means-Ends Analysis," Unpublished Manuscript, February 2003.

[8] Schwartau W., Information *Warfare*, 2nd ed., p. 12, Thunder's Mouth Press, October 1996.

[9] Denning D.E., *Information Warfare and Security*, pp. 3-19, Addison Wesley, February 2001.

[10] CERT Advisory CA-2003-04 MS-SQL Server Worm, http://www.cert.org/advisories/CA-2003-04.html, February 2003.

[11] Debar H., Dacier M., Wespi A., "Towards a taxonomy of intrusion-detection systems," *Computer Networks*, v. 31, no. 8, pp. 805-822, April 1999.

[12] Denning D.E., "An Intrusion-Detection Model," *IEEE Trans. Software Eng.*, v. SE-13, no. 2, pp. 222-232, February 1987.

[13] Lindqvist, U., Porras, A.P., "Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST)," *in Proc. the 1999 IEEE Symposium on Security and Privacy*, Oakland, California, May 9–12, 1999.

[14] Ko, C., Ruschitzka, M., Levitt, "Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach," *in Proc. the 1997 IEEE Symposium on Security and Privacy*, Oakland, California, May 4–7, 1997.

[15] Phung, M., "Data Mining in Intrusion Detection", http://www.sans.org/resources/idfaq/data_mining.php, February 2003.

[16] Brackney, R., "Cyber-Intrusion Response," *in Proc. 17th IEEE Symposium on Reliable Distribution Systems,* West Lafayette, Indianapolis, October 20-23, 1990.

[17] Petkac, M., Badger, L., "Security Agility in Response to Intrusion Detection", *in Proc. of the 16th Annual Computer Security Applications Conference*, New Orleans, Louisiana, pp. 11-20, December 11-15, 2000.

[18] Somayaji, A. B., *Operating System Stability and Security through Process Homeostasis,* Ph.D. Dissertation, University of New Mexico, 2002.

[19] Lewandowski, S., Van Hook, D., O'Leary, G., Haines, J., Rossey, L., "SARA: Survivable Autonomic Response Architecture," *in Proc. DARPA Information Survivability Conference & Exposition II*, pp. 77-88, vol.1, June 2001.

[20] Sekar, R., Bowen, T., Segal, M., "On Preventing Intrusions by Process Behavior Monitoring," USENIX Intrusion Detection Workshop, Santa Clara, California, pp. 29-40, April 1999.

[21] Balasubramaniyan, J., Garcia-Fernandez, J., Isacoff, D., Spafford, E., Zamboni, D., *An Architecture for Intrusion Detection using Autonomous Agents*, Technical Report TR-98-05, Purdue University, June 1998.

[22] IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Std. 1471-2000, September 21, 2000.

[23] Kruchten, P., "Architectural Blueprints—The "4+1" View Model of Software Architecture," *IEEE Software*, v. 12, no. 6, pp. 42-50, November 1995.

[24] Hofmeister, C., Nord, R., and Soni, D., *Applied Software Architecture*, Addison Wesley, April 2001.

[25] Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language User Guide*, Addison Wesley, October 1998.

[26] Shaw, M. and Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, April 1996.

[27] Garlan, D. and Shaw, M., "An Introduction to Software Architecture," in *Advances in Software Engineering and Knowledge Engineering*, v. 2, pp. 1-39, New York, New York, 1993.

[28] Allen, R. J., *A Formal Approach to Software Architecture*, Ph.D. Thesis, Carnegie Mellon University, Technical Report CMU-CS-97-144, May 1997.

[29] Architecture Description Languages, Carnegie Mellon University, Software Engineering Institute, http://www.sei.cmu.edu/architecture/adl.html, February 2003.

[30] Hilliard, R., "Using the UML for Architectural Description," *in Proc. UML'99 The Unified Modeling Language, Second International Conference*, Lecture Notes in Computer Science volume 1723, Springer, 1999.

[31] Hofmeister, C., Nord, R. L., and Soni, D., "Describing Software Architecture with UML", *in Proc. 1st Working IFIP Conference on Software Architecture (WICSA1),* pp. 145-159, San Antonio, Texas, February 1999.

[32] Ko, C., Fraser, T., Badger, L., Kilpatrick, D., "Detecting and Countering System Intrusions Using Software Wrappers", 9th USENIX Security Symposium, Denver, CO, pp. 145-156, August 2000.

[33] Fraser, T., Badger, L., and Feldman, M., "Hardening COTS Software with Generic Software Wrappers," *in Proc. IEEE Symposium on Security and Privacy*, pp. 2-16, May 1999.

[34] Holling, C. S., "Engineering Resilience vs. Ecological Resilience," *Engineering Within Ecological Constraints*, National Academy Press, Washington, D.C, pp. 32-43, 1996.

[35] Jain, R., *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, April 1991.

[36] P.Oreizy *et al.,* "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems*, v.14, no.3, pp. 54-62, 1999.

[37] Westbrook, A., Ratti, O., *Aikido and the Dynamic Sphere*, Charles E. Tuttle Co., September 1994.

[38] Ellison, R.J. *et al.*, "Survivable Network System Analysis: A Case Study," *IEEE Software*, v. 16, no. 4, pp. 70-77, July/August 1999.

[39] Kruchten, P., *The Rational Unified Process: An Introduction*, 2nd ed., Addison Wesley, March 2000.

[40] Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, 1st ed., Prentice-Hall, 1998.

[41] Garlan, D., "What is Style?" *in Proc. Dagshtul Workshop on Software Architecture*, Saarbruecken, Germany, February 1995.

[42] Widespread exploitation of *rpc.statd* and *wu-ftpd* vulnerabilities. Incident note IN-2000-10, CERT Coordination Center, Carnegie Mellon University, Pittsburgh, Pennsylvania, September 15, 2000.

[43] Hofmeister, C., Nord, R., and Soni, D., "Software Architecture in Industrial Applications," *in Proc. 17th International Conference on Software Engineering*, Seattle, Washington, pp. 196-207, April 1995.

DISTRIBUTION LIST

1. Defense Technical Information Center
   Fort Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. J. Bret Michael
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California

4. Richard Riehle
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California

5. Deniz Kuvvetleri Komutanligi (Turkish Navy Headquarters)
   Ankara, Turkey

6. Deniz Harp Okulu Komutanligi (Turkish Naval Academy)
   Istanbul, Turkey

7. Arastirma Merkezi Komutanligi
   Yazilim Gelistirme Grup Baskanligi
   (Turkish Navy Software Development Center)
   Istanbul, Turkey